LESLIE HARLLEY WATTER

L@V@ UM SIMULADOR PARA O MICROCONTROLADOR AVR8515

Trabalho de Graduação apresentado ao Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2003

Agradecimentos

À minha família, pelo apoio dado, sem o qual eu não teria conseguido.

Ao prof. Roberto A. Hexsel, por ter acreditado em mim.

Aos amigos Nilson Bastos Júnior, que me fez ver aplicações que deram um novo ânimo durante o processo de desenvolvimento, Aristeu S. Rozanski Filho e Eduardo Pereira Habkost pelas discussões sobre o simulador, e Fernando M. Roxo da Motta que, com aplicações da física me fez ver a importância de um trabalho como esse.

Sumário

Lista de Figuras iv

Lista de Tabelas v

1 Introdução 1

2 O Microcontrolador AVR8515 3

2.1 Funcionalidades . . . 3
2.2 Descrição da Arquitetura . . . 4
2.3 PC (Contador de Programa) . . . 7
2.4 Apontador de Pilha . . . 8
2.5 Registradores de Uso Geral . . . 8
2.5.1 Registradores X, Y e Z . . . 9
2.6 ULA (Unidade Lógico-Aritmética) . . . 9
2.7 Registrador de Status . . . 9
2.8 Memória de Programa . . . 10
2.9 SRAM . . . 11
2.9.1 Modos de Endereçamento de Programa e de Dados . . . 12
2.10 Endereçamento de E/S . . . 18
2.11 Portas de E/S (A,B,C,D) . . . 20
2.11.1 PortA (PA7..PA0) . . . 20
2.11.2 PortB (PB7..PB7) . . . 20
2.11.3 PortC (PC7..PC0) . . . 21
2.11.4 PortD (PD7..PD0) . . . 21
2.12 Registrador de Controle Mestre . . . 21
2.13 Comparador Analógico . . . 22
2.14 UART . . . 22
2.15 Interface Serial de Periféricos . . . 23
2.16 EEPROM . . . 23
2.17 Unidade de Interrupções . . . 23
2.18 Temporizadores/Contadores . . . 24
2.19 Watchdog Timer . . . 24

3 Programação do AVR8515 **25**
3.1 Conjunto de Instruções do AVR8515 25
3.1.1 Nomenclatura utilizada 25
3.1.2 Conjunto de Instruções 26
3.2 Material Adicional 31

4 L@V@ - Um Simulador para o 8515 **32**
4.1 Descrição Geral do Simulador 32
4.1.1 Interface do Simulador 32
4.1.2 Fluxo de execução no Simulador 34
4.2 Estruturas utilizadas para implementar o simulador 35
4.2.1 PC 36
4.2.2 Implementação das Instruções 36
4.2.3 Apontador da Pilha / Stack Pointer 37
4.2.4 Registradores de Uso Geral 37
4.2.5 Registrador de Status 38
4.2.6 Memória de Programas 39
4.2.7 Memória de Dados 39
4.2.8 SRAM 40
4.2.9 Memória de E/S 40
4.3 Funções do Simulador 40
4.3.1 Função *read_hexa()* 40
4.3.2 Função *decodificador_instrucao()* 42
4.3.3 Função *execution_step()* 42
4.3.4 Depuração 45
4.4 Requisitos do Simulador 45

5 Conclusão **46**

A Instruções Reconhecidas **48**

Referências Bibliográficas **53**

Lista de Figuras

2.1 Diagrama do Hardware do AVR8515 5
2.2 Organização da Memória do AVR8515 7
2.3 Registradores de Uso Geral 8
2.4 Organização dos Registradores Duplos 9
2.5 Registrador de Status 9
2.6 Organização da Memória SRAM 12
2.7 Endereçamento Direto com Um Registrador 13
2.8 Endereçamento Direto com Dois Registradores 13
2.9 Endereçamento Direto da Memória de E/S 14
2.10 Endereçamento Direto da Memória de Dados 14
2.11 Endereçamento Indireto da Memória de Dados com Deslocamento 15
2.12 Endereçamento Indireto da Memória de Dados 15
2.13 Endereçamento Indireto da Memória de Dados com Pré-Decremento 16
2.14 Endereçamento Indireto da Memória de Dados com Pós-Incremento 16
2.15 Endereçamento Constante da Memória de Programa 17
2.16 Endereçamento Indireto da Memória de Programa 17
2.17 Endereçamento Relativo da Memória de Programa 18

4.1 Interface do Simulador 33

Lista de Tabelas

2.1 Endereços de Memória de E/S 19
2.2 Vetor de Interrupções 24

3.1 Instruções Lógicas e Aritméticas 27
3.2 Instruções de Desvio 28
3.3 Instruções de Transferência de Dados 29
3.4 Instruções de Manipulação de Bits 30

A.1 Instruções Absolutas, nenhum bit é modificável 48
A.2 Instruções que tem a máscara de bits modificáveis 0x00F0 49
A.3 Instruções que tem a máscara de bits modificáveis 0x0077 49
A.4 Instruções que tem a máscara de bits modificáveis 0x00FF 49
A.5 Instruções que tem a máscara de bits modificáveis 0x00F1 49
A.6 Instruções que tem a máscara de bits modificáveis 0x01F0 50
A.7 Instruções que tem a máscara de bits modificáveis 0x01F7 50
A.8 Instruções que tem a máscara de bits modificáveis 0x03F8 51
A.9 Instruções que tem a máscara de bits modificáveis 0x03FF 51
A.10 Instruções que tem a máscara de bits modificáveis 0x07FF 52
A.11 Instruções que tem a máscara de bits modificáveis 0x0FFF 52

Capítulo 1

Introdução

Um computador consiste basicamente de três componentes principais: a unidade central de processamento (CPU), memória de dados e programa e sistema de entrada e saída (E/S). A CPU controla o fluxo de informações entre os componentes do computador e também processa os dados através de operações. A maior parte do processamento é executada na unidade lógica e aritmética (ULA/ALU) dentro da CPU. Quando a CPU de um computador é construída em uma única placa de circuitos, o computador é chamado de minicomputador. Um microprocessador é o nome dado à uma CPU colocada em um único chip. Microprocessadores são dispositivos de uso geral, apropriados para diversas aplicações. Um computador criado em torno de um microprocessador é chamado de microcomputador. A escolha do sistema de Entrada/Saída de um microcomputador irá depender de sua aplicação específica, por exemplo, a maioria dos computadores pessoais possui um teclado e um monitor como dispositivos de entrada e saída padrão [5].

Um microcontrolador é um microcomputador completo construído em um único chip. Os sistemas de E/S e de memória contidos em um microcontrolador especializam estes dispositivos para que eles possam se comunicar/interfacear com outros hardwares e também com funções de controle de aplicações. Uma vez que os microcontroladores são equipados com poderosos processadores digitais, o grau de controle e programabilidade que eles provêm aumenta significativamente a efetividade da aplicação. Por exemplo, microcontroladores são utilizados como controladores do motor nos automóveis e no controle de foco e de exposição nas máquinas fotográficas. Para atenderem aos requisitos funcionais destas aplicações, eles tem uma alta concentração de componentes no próprio circuito integrado, como portas seriais, portas paralelas de entrada e saída, temporizadores, contadores, controle de interrupções, conversores analógico-digital, memória de escrita/leitura (RAM) e memória somente de leitura(ROM).

Aplicações de controle de sistemas embarcados também distingüem microcontroladores de microprocessadores de uso geral. Sistemas embarcados freqüentemente necessitam de operações de tempo real e multitarefa. Operação de tempo real implica que o controlador em-

barcado precisa ser capaz de receber e processar os sinais do seu ambiente quando eles estão disponíveis, isto é, o ambiente não deve esperar que o controlador fique disponível. Da mesma maneira, o controlador precisa executar rápido o suficiente para emitir sinais de controle para o seu ambiente quando eles se tornam necessários. Novamente o ambiente não deve esperar pelo controlador. Em outras palavras, o controlador embarcado não pode ser o gargalo da operação do sistema. Operação multitarefa é a capacidade de executar várias funções de uma maneira simultânea ou quase-simultânea. O controlador embarcado freqüentemente é responsável por monitorar diversos aspectos de um sistema e deve ser capaz de responder apropriadamente quando a necessidade surge.

Este trabalho refere-se ao AVR8515 [1] porque este é o microcontrolador empregado nos laboratórios do Bacharelado em Ciência da Computação e didaticamente interessante por ser regular e da ortogonalidade[1] no uso dos recursos. O AVR8515 foi escolhido também pelo fato de ser largamente utilizado por profissionais da área, tendo um público alvo grande e acostumado com a utilização deste microcontrolador em trabalhos dos mais variados. O fabricante recomenda o uso do mega8515 porque este opera com velocidades de relógio mais elevadas e possui uma porta de E/S adicional (porta E).

Simulador

O L@V@ tem como objetivo principal auxiliar no aprendizado, fornecendo uma ferramenta que se aproxima o máximo possível do hardware, fazendo com que o aprendizado se torne mais rápido uma vez que é feita a associação entre o conhecimento já adquirido na teoria, através de explanações em sala de aula, e o conhecimento a ser adquirido na prática, pelo acompanhamento passo a passo da execução dos programas. A depuração dos programas é muito mais simples de ser feita no L@V@ do que em um sistema de hardware/software. O simples fato de acompanhar as instruções e suas modificações no momento em que são executadas facilita a criação do fluxo de execução do programa em uma visão mental que possibilita a visualização de correções e comparação com o resultado desejado tanto no “fim” do processamento quanto em uma parte específica.

Para o profissional, o L@V@ traz benefícios tais como: maior rapidez no treinamento (durante a fase de aprendizagem), e maiores eficiência, rapidez e qualidade na produção do código, uma vez que eventuais erros podem ser facilmente detectados e corrigidos. Ele também permite a redução dos custos de desenvolvimento, possibilitando efetuar os testes de código no simulador e, somente quando o código for considerado estável, enviá-lo à fase de testes no hardware.

[1]regularidade.

Capítulo 2

O Microcontrolador AVR8515

O AVR8515, ou simplesmente 8515, é um processador segmentado *com um pipeline de 2 estágios*, que utiliza a arquitetura de Harward [3]. A busca de instruções e acesso à memória de dados é feita através de duas portas, uma para instruções (16bits) e outra para dados (8bits). O 8515 é um processador RISC, com codificação das instruções (quase-)regular e modos de endereçamento relativamente simples. Observaremos a seguinte sintaxe durante este documento: \$20 $\equiv$ hexadecimal 0x20, AVR8515 $\equiv$ 8515.

2.1 Funcionalidades

Nesta seção são apresentados os componentes de hardware do 8515, objetivando fornecer uma referência para que o leitor possa compreender as funcionalidades do microcontrolador e entender o funcionamento do simulador. O microcontrolador 8515 possui os seguintes componentes:

- 32 Registradores de 8 bits
- 32 Linhas de E/S de Uso Geral
- 64 Registradores de E/S
- 8 Kbytes de Memória Flash Re-programável
- 512 bytes de RAM Interna
- 512 bytes de Memória EEPROM
- 1 Temporizador/Contador de 8 bits com Divisor de Freqüências Separado
- 1 Temporizador/Contador de 16 bits com Divisor de Freqüências Separado

- 1 Comparador Analógico
- 1 Watchdog Programável com Oscilador Interno
- 1 UART Programável
- 1 Interface Serial de Periféricos (Serial Peripheral Interface - SPI)
- 2 Modos Selecionáveis de Economia de Energia
- 12 Fontes de Interrupções Diferentes

A família de produtos do 8515 possui diferentes tipos de empacotamento, entre os extremos de versão com 40 pinos (8515) e versão com 8 pinos (2343). As versões reduzidas são baseadas no mesmo núcleo mas dispõe de menos recursos acessíveis externamente.

A figura 2.1 mostra a organização do 8515.

2.2 Descrição da Arquitetura

Esta seção tem como objetivo fornecer uma visão geral da arquitetura do microcontrolador 8515. As informações aqui descritas serão explicadas em mais detalhes nas seções a seguir.

O microcontrolador 8515 utiliza o conceito de arquitetura de Harward, com barramentos e memórias separados para a busca das instruções (EEPROM) e para acesso aos dados (RAM) e registradores de E/S.

Os programas são executados em um pipeline de 2 estágios que, enquanto uma instrução é executada, a próxima instrução é buscada antecipadamente da memória de programas. Esta implementação permite que existam até duas instruções executando a cada ciclo de clock do microcontrolador.

O bloco de registradores (32 registradores de 8 bits cada) é acessado de maneira rápida e fácil, e como está diretamente conectado com a ULA (Unidade Lógica e Aritmética), permite que operações com os registradores possam ser executadas em um único ciclo de clock. Desta maneira, em um único ciclo, os registradores são acessados, a operação na ULA é executada e o resultado é armazenado novamente nos registradores. [Veja a seção registradores em 2.5, e a seção ULA em 2.6 para mais detalhes]

Seis dos registradores presentes no bloco de registradores (de R26 até R31) podem ser utilizados como três registradores duplos para o endereçamento indireto de dados. Estes registradores duplos são chamados X, Y e Z, e possuem grande importância no acesso aos dados da memória RAM, tanto a interna como a externa. A memória RAM interna tem um tamanho de

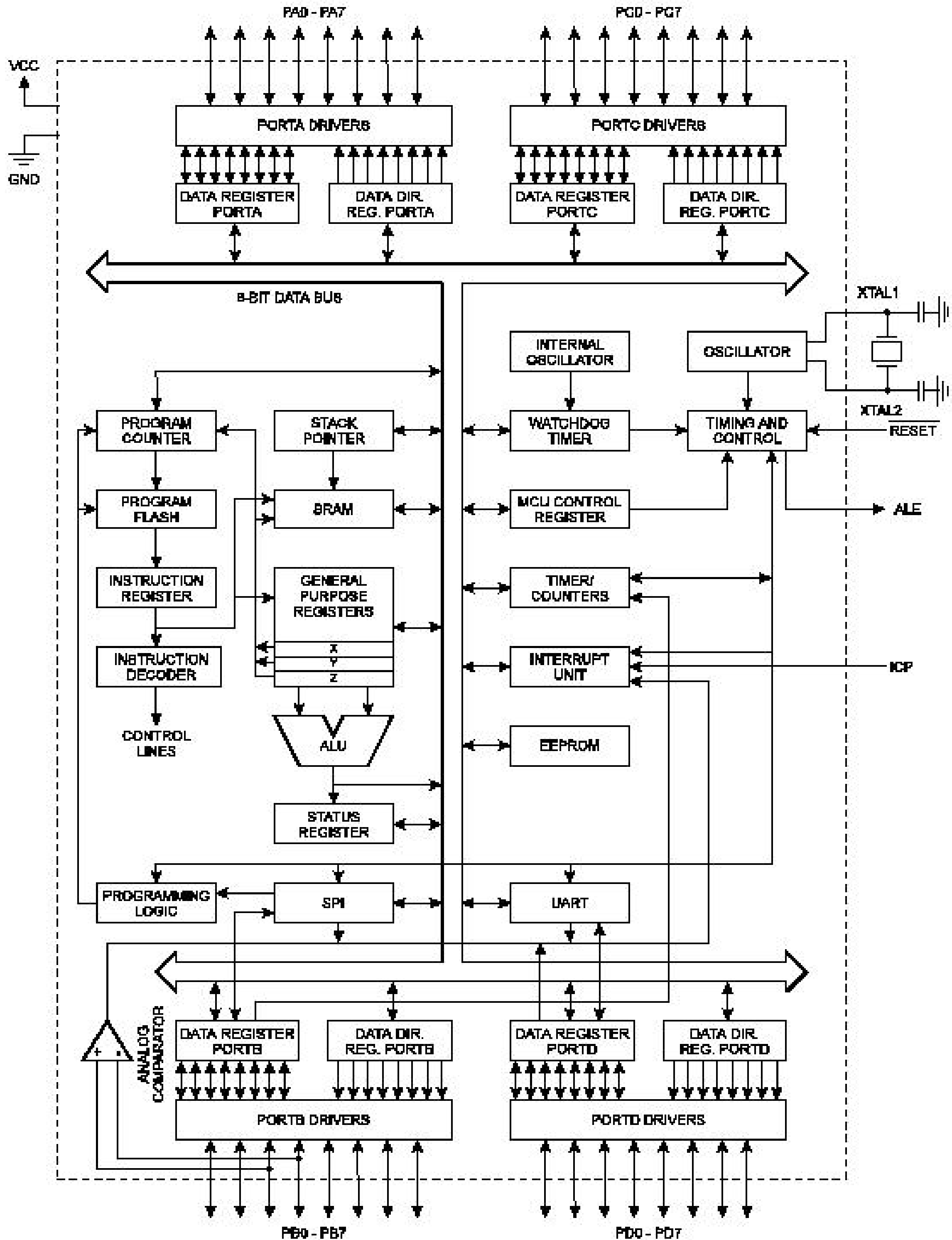

Figura 2.1: *Diagrama do Hardware do AVR8515*

512 bytes e pode ser acessada através de um dos seguintes métodos de endereçamento: [Veja a seção SRAM em 2.9]

1. endereçamento direto

2. endereçamento indireto
3. endereçamento indireto com deslocamento
4. endereçamento indireto com pré-decremento
5. endereçamento indireto com pós-incremento

Também é possível utilizar-se dos mesmos modos de endereçamento para acessar os registradores de uso geral. Porque estes registradores estão mapeados nas 32 primeiras posições do espaço de dados (de $00 até $1F), ou seja, é possível manipular os registradores como se fossem parte da memória RAM.

Observando o espaço de endereçamento de E/S, vemos que este contêm 64 posições de acesso aos registradores dos periféricos e de controle da própria CPU. A memória de E/S pode ser acessada diretamente através dos métodos já mencionados, ou também como sendo uma continuação da memória RAM, com posições seguintes às do bloco de registradores ($20 - $5F). Podemos ver essas faixas de endereços mais claramente na figura 2.2.

Observe que a memória de programas é organizada em um bloco de 4Kx16, uma vez que a grande maioria das instruções do 8515 possui 16 bits de largura, facilitando assim o alinhamento das instruções na memória e tornando o sistema mais eficiente.

Cada endereço de programa contém uma instrução de 16 ou 32 bits. Utilizando as instruções "jump relativo" (jump) e "chamada de rotina" (call), o espaço de 4K endereços pode ser acessado diretamente.

Durante interrupções e chamadas de subrotina, o endereço de retorno do contador de programa (PC) é armazenado na pilha, que é alocada na SRAM (e conseqüentemente é limitada pelo tamanho e utilização total da SRAM). Todos os programas devem inicializar o apontador da pilha (SP) na rotina de *reset* e antes da execução de subrotinas ou interrupções. [Veja detallhes do PC na seção 2.3 e do SP na seção 2.4]

Cada interrupção diferente tem um vetor de interrupções separado na tabela de vetores de interrupção no início da memória de programas, sendo que cada uma delas possui uma prioridade diferente de acordo com sua posição no vetor de interrupções (quanto menor o endereço, maior a prioridade). [Vetor interrupções 2.17] O registrador de status tem um bit que controla se as interrupções são habilitadas ou desabilitadas globalmente, independendo de suas configurações individuais. [Veja a seção 2.7 para detalhes do Registrador de Status]

Existem ainda 4 portas de E/S que tem múltiplas funções, como, por exemplo, fornecer um meio de acesso à memória SRAM externa, controlar os temporizadores/contadores e também fornecer uma interface para comunicação em modo serial e/ou paralelo. [Veja a seção 2.11 para detalhes das Portas de E/S]

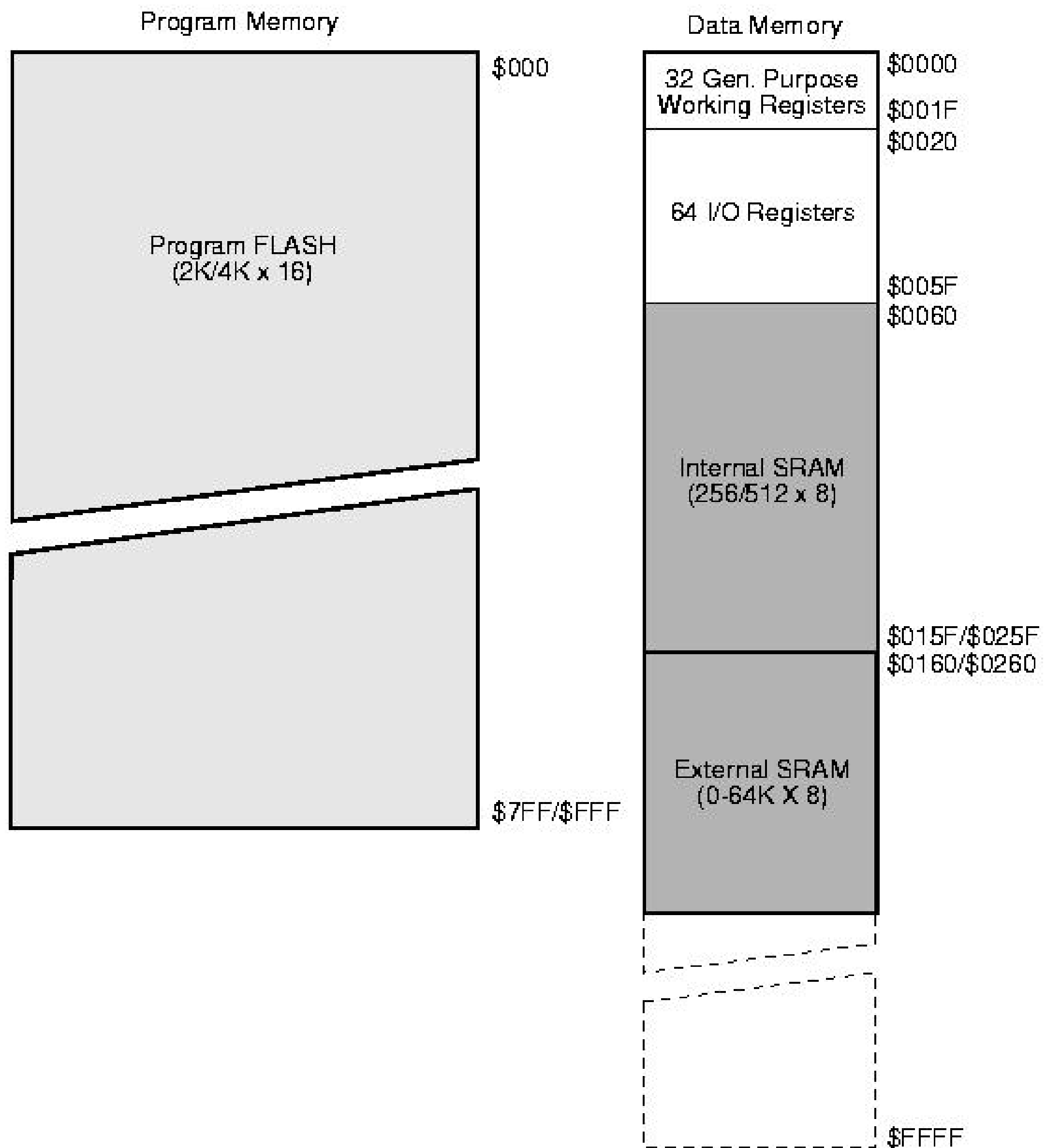

Figura 2.2: *Organização da Memória do AVR8515*

2.3 PC (Contador de Programa)

O contador de programa (PC) do 8515 tem 12 bits de tamanho, suficiente para endereçar as 4096 posições da memória de programa. Apesar de as instruções poderem gerar valores de PC com 16 bits, somente 12 bits são utilizados nesta implementação.

2.4 Apontador de Pilha

O apontador da pilha (*Stack Pointer*) do 8515 é de 16 bits, constituído por dois registradores ($3E e $3D) de 8 bits no espaço de E/S. Como o 8515 suporta até 64Kb de SRAM externa, todos os 16 bits são utilizados para indexar a pilha.

2.5 Registradores de Uso Geral

A figura 2.3 mostra a estrutura dos 32 registradores de uso geral na CPU.

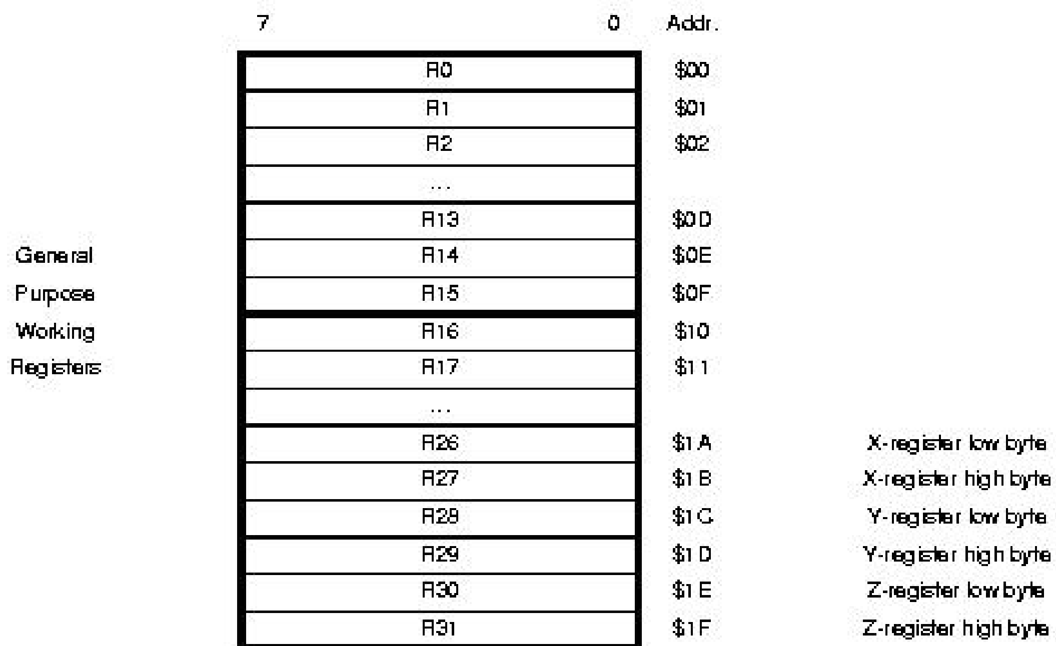

Figura 2.3: *Registradores de Uso Geral*

Todas as instruções que operam com os registradores executam em um único ciclo de clock. As únicas exceções são as cinco instruções aritméticas e lógicas que operam com uma constante e um registrador (SBCI, SUBI, CPI, ANDI e ORI) e a instrução LDI para loads imediatos. Estas instruções se aplicam na segunda metade dos registradores no bloco de registradores, de R16 a R31. As instruções SBC, SUB, CP, AND e OR e todas as outras instruções/operações que utilizam dois registradores ou mesmo aquelas que utilizam um único registrador se aplicam a todo o bloco de registradores.

Como mostrado na figura 2.2, cada registrador é também mapeado como um endereço da memória de dados, mapeando-os diretamente nas primeiras 32 posições do espaço de dados. Embora não implementados fisicamente como posições da SRAM, esta organização de memória provê grande flexibilidade no acesso aos registradores. Valendo-se dessa característica, os registradores X,Y e Z podem indexar qualquer registrador no bloco.

2.5.1 Registradores X, Y e Z

Os registradores de R26 até R31 tem algumas funções a mais que os registradores de uso geral. Estes registradores são ponteiros de endereço para endereçamento indireto do espaço de dados. Os três registradores de endereçamento indireto X, Y e Z são assim definidos:

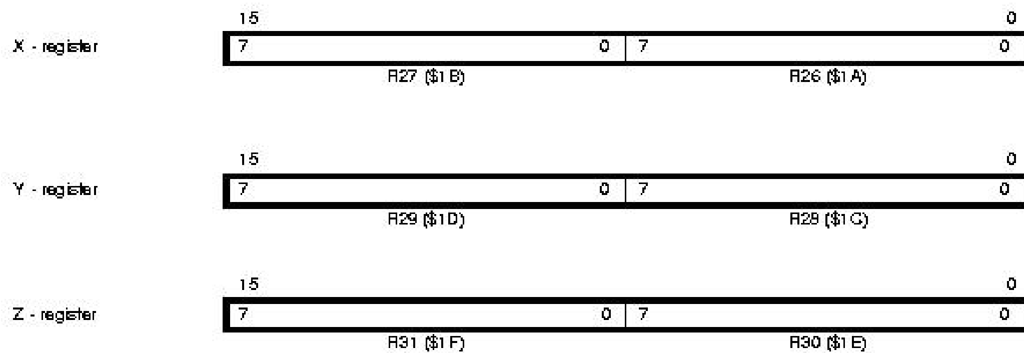

Figura 2.4: *Organização dos Registradores Duplos*

Nos diferentes modos de endereçamento, estes registradores de endereçamento tem funções como deslocamento fixo e incremento ou decremento automáticos, variando de acordo com a instrução.

2.6 ULA (Unidade Lógico-Aritmética)

A ULA opera conectada diretamente com os 32 registradores de uso geral. As operações da ULA são divididas em três grandes categorias - operações aritméticas, operações lógicas e operações com bits.

2.7 Registrador de Status

O registrador de status do 8515 (alocado no endereço de E/S $3F ($5F)), é definido como a seguir:

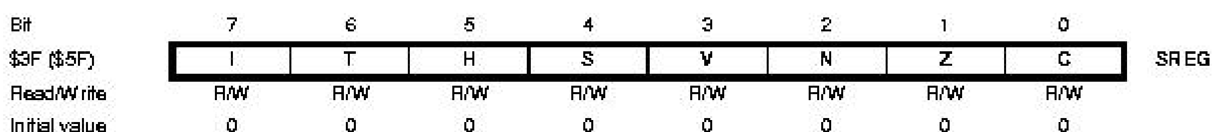

Figura 2.5: *Registrador de Status*

Bit 7 - I: Global Interrupt Enable Este bit deve estar setado ("em um") para que as interrupções sejam habilitadas. Se o bit I estiver zerado ("em zero"), nenhuma das interrupções estará habilitada, independente das configurações individuais de cada interrupção. O bit I é zerado pelo hardware após a ocorrência de uma interrupção, e setado ("em um") pela instrução RETI para habilitar as interrupções subseqüentes.

Bit 6 - T: Bit Copy Storage As instruções de cópia de bits BLD (Bit LoaD) e BST (Bit STore) utilizam o bit T como fonte e destino para o bit operado. Um bit de um registrador do bloco de registradores pode ser copiado para T pela instrução BST, e um bit em T pode ser copiado para um bit em um registrador no bloco de registradores pela instrução BLD.

Bit 5 - H: Half Carry Flag Este bit contém o vai-um entre dois dígitos hexadecimais (do bit 3 para o bit 4) em algumas operações aritméticas.

Bit 4 - S: Sign Bit, S = N $\oplus$ V O bit S é sempre o resultado de um *ou exclusivo* entre o bit de sinal negativo e o bit de sinal de overflow (bit 3).

Bit 3 - V: Two's Complement Overflow Flag O bit de sinal V é utilizado na aritmética de complemento de dois.

Bit 2 - N: Negative Flag O bit de sinal N indica um resultado negativo após uma operação aritmética ou lógica.

Bit 1 - Z: Zero Flag O bit de sinal Z indica que o resultado após uma operação aritmética ou lógica é zero.

Bit 0 - C: Carry Flag O bit de sinal C indica que há a presença de *vai-um* (carry) em uma operação aritmética ou lógica.

Observe que o registrador de status *não é salvado ou restaurado automaticamente* quando ocorre a entrada ou a saída de uma rotina de tratamento de interrupção.

2.8 Memória de Programa

O 8515 contêm memória flash com capacidade de armazenamento de 8K bytes ou 4K instruções. Uma vez que todas as instruções possuem 16 ou 32 bits de tamanho, a memória flash é organizada como 4Kx16, facilitando o alinhamento das instruções na memória e garantindo um melhor desempenho. A memória flash tem uma durabilidade garantida de pelo menos 1000 ciclos de escrita.

2.9 SRAM

A figura 2.6 mostra o espaço de endereçamento de dados e periféricos. As 608 posições mais baixas da memória de dados endereçam o bloco de registradores, a memória de E/S e a SRAM interna. As primeiras 96 posições endereçam o bloco de registradores mais a memória de E/S, e as próximas 512 posições endereçam a SRAM interna. Uma SRAM externa (opcional) pode ser inserida no mesmo espaço de memória. Esta SRAM irá ocupar a próxima localização a partir da última posição da SRAM interna até o limite de 64K - 1, dependendo do tamanho da SRAM externa utilizada.

Quando os endereços acessando o espaço de memória de dados ultrapassarem o número de posições da SRAM interna, o conteúdo da SRAM externa será acessado usando as mesmas instruções que um acesso à SRAM interna. Quando o acesso é feito à área interna do espaço de dados, os pinos de "read" e "write" estarão inativos durante todo o ciclo de acesso. O acesso à SRAM externa é habilitado mudando o valor do bit SRE para 1 no registrador de controle mestre (MCUCR)[Veja mais detalhes sobre o MCUCR na seção 2.12].

O acesso à SRAM externa demora um ciclo adicional por byte comparado ao acesso à SRAM interna. Isso significa que as instruções LD, ST, LDS, STS, PUSH e POP irão demorar um ciclo adicional durante sua execução. Se a pilha for localizada em uma SRAM externa, as interrupções, chamadas de subrotina e retornos de chamada irão demorar dois ciclos extras porque um PC de dois bytes será empilhado e desempilhado.

Os diferentes modos de endereçamento para a memória de dados são: direto, indireto, indireto com deslocamento, indireto com pré-decremento e indireto com pós-decremento. No bloco de registradores, os registradores de R26 até R31 também são apontadores para endereçamento indireto de memória.

O endereçamento direto atinge todo o espaço de dados. O modo de endereçamento indireto com deslocamento permite deslocamentos de até $\pm$ 32 posições com relação ao endereço base. Quando utilizamos um registrador no modo de endereçamento indireto com pré-decremento automático e pós-incremento, os registradores de endereço X,Y e Z são decrementados e incrementados, respectivamente.

Os 32 registradores de uso geral, 64 registradores de E/S, os 512 bytes de dados da SRAM interna, e os 64K bytes de dados da SRAM externa opcional no 8515 são acessíveis através de todos esses modos de endereçamento.

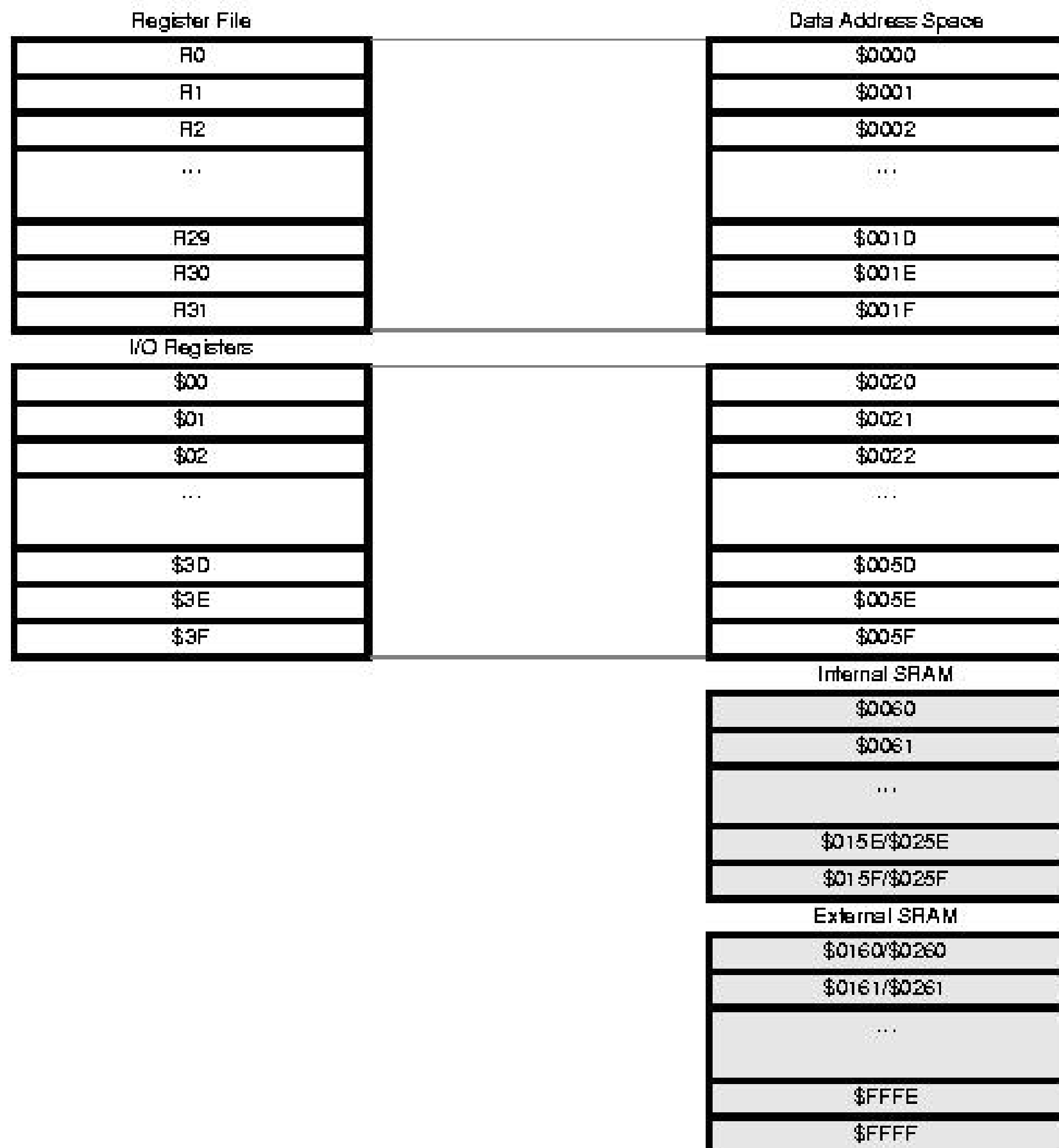

Figura 2.6: *Organização da Memória SRAM*

2.9.1 Modos de Endereçamento de Programa e de Dados

Esta seção descreve os diferentes modos de endereçamento suportados pela arquitetura AVR. Nas figuras, OP significa a parte do opcode[1] da instrução. Para simplificar as figuras, a exata localização dos bits de endereço não será mostrada porque a codificação das instruções é irregular.

[1]código.

Endereçamento Direto com Um Registrador

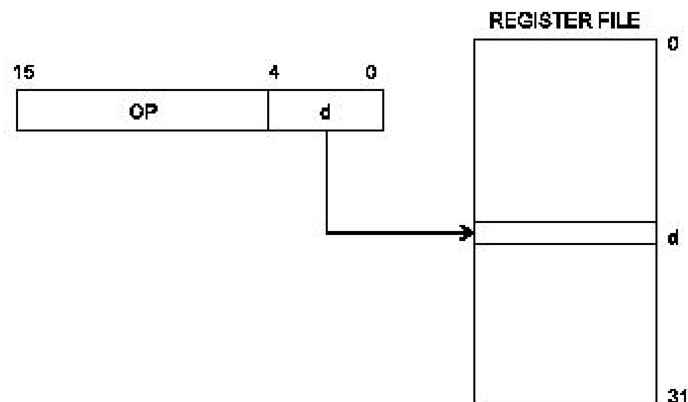

Figura 2.7: *Endereçamento Direto com Um Registrador*

O operando está contido no registrador destino d (Rd).

Exemplo: NEG r2 *(r2 $\leftarrow$ $00 - r2[2])*

Endereçamento Direto com Dois Registradores

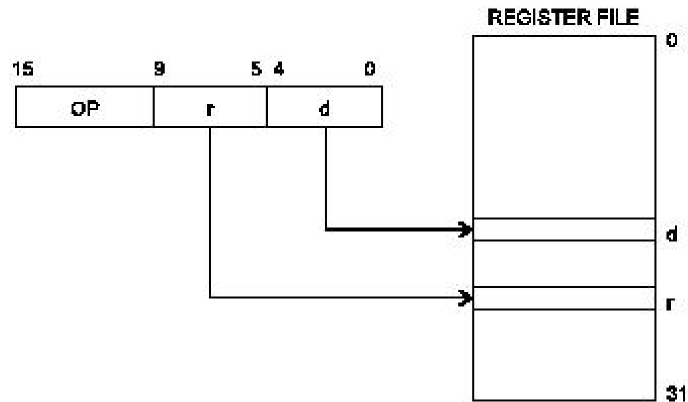

Figura 2.8: *Endereçamento Direto com Dois Registradores*

Os operandos estão nos registradores r (Rr) e d (Rd). O resultado é armazenado em d (Rd).

Exemplo: ADD r2, r3 *(r2 $\leftarrow$ r2 + r3)*

[2]Notação: r2 recebe o valor de $00 menos o conteúdo de r2

Endereçamento Direto da Memória de E/S

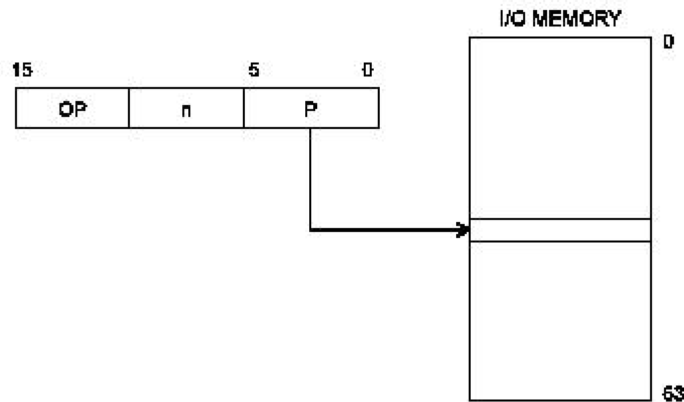

Figura 2.9: *Endereçamento Direto da Memória de E/S*

O endereço do operando está contido nos 6 bits da palavra da instrução. n é o endereço do registrador fonte ou de destino.

Exemplo: IN r2, $16 *(r2 ← E/S($16))*

Endereçamento Direto da Memória de Dados

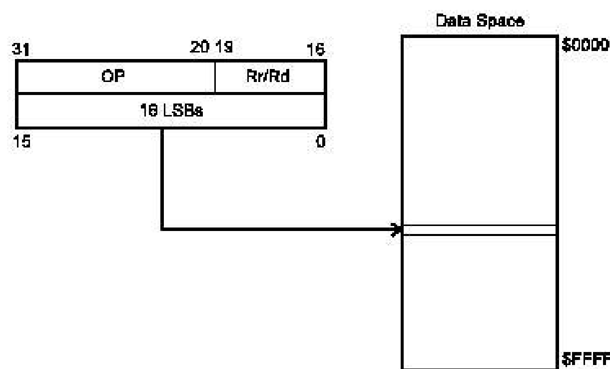

Figura 2.10: *Endereçamento Direto da Memória de Dados*

Um endereço de 16-bits está contido nos 16LSBs[3] de uma instrução com duas palavras. Rr/Rd especificam o registrador fonte ou destino.

Exemplo: LDS r2, $FF00 *(r2 ← ($FF00))*

[3]LSBs (Least Significant Bits)

Endereçamento Indireto da Memória de Dados com Deslocamento

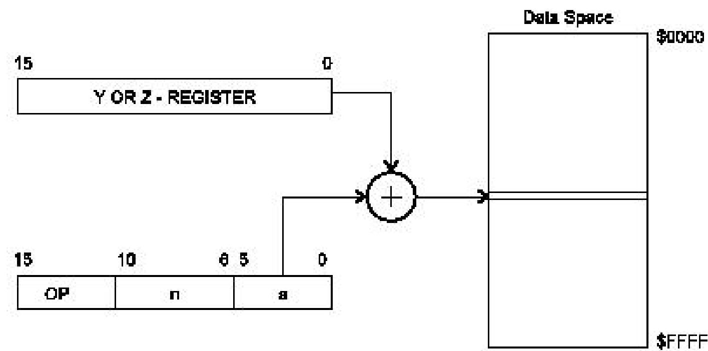

Figura 2.11: *Endereçamento Indireto da Memória de Dados com Deslocamento*

O endereço é o resultado do conteúdo do registrador Y ou Z adicionado ao endereço contido nos 6 bits da instrução.

Exemplo: LD r2, Y+q *(r2 ← (Y+q))*

Endereçamento Indireto da Memória de Dados

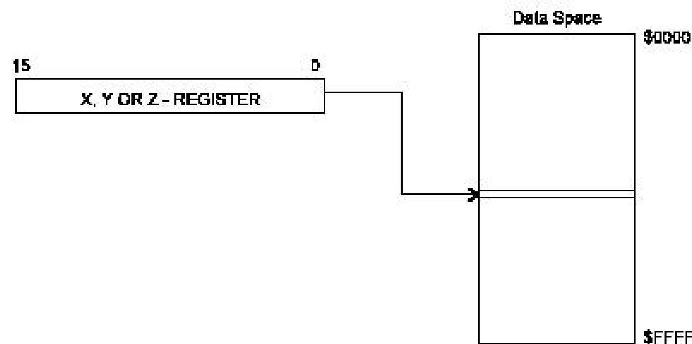

Figura 2.12: *Endereçamento Indireto da Memória de Dados*

O endereço do operando é o conteúdo do registrador X, Y ou Z.

Exemplo: LD r2, X *(r2 ← (X))*

Endereçamento Indireto da Memória de Dados com Pré-Decremento

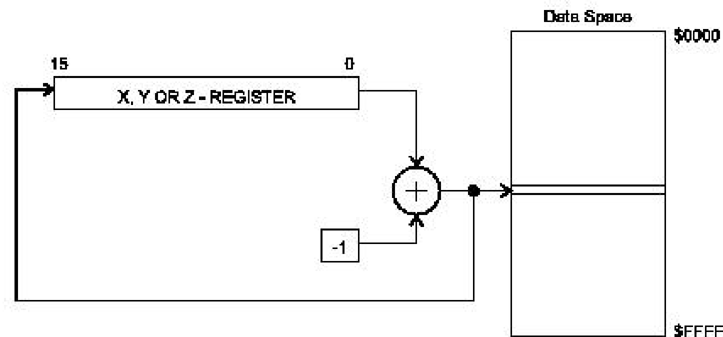

Figura 2.13: *Endereçamento Indireto da Memória de Dados com Pré-Decremento*

O registrador X, Y ou Z é decrementado antes da operação. O endereço do operando é o conteúdo do registrador X, Y ou Z decrementado.

Exemplo: LD r2, -X *(X ← X - 1;r2 ← (X))*

Endereçamento Indireto da Memória de Dados com Pós-Incremento

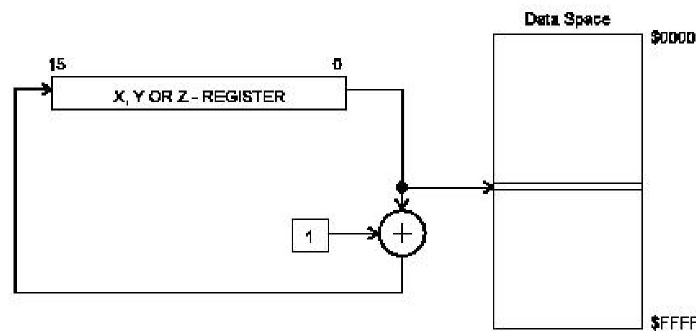

Figura 2.14: *Endereçamento Indireto da Memória de Dados com Pós-Incremento*

O registrador X, Y ou Z é incrementado após a operação. O endereço do operando é o conteúdo do registrador X, Y ou Z antes do incremento.

Exemplo: LD r2, X+ *(r2 ← (X); X ← X + 1)*

Endereçamento Constante da Memória de Programa

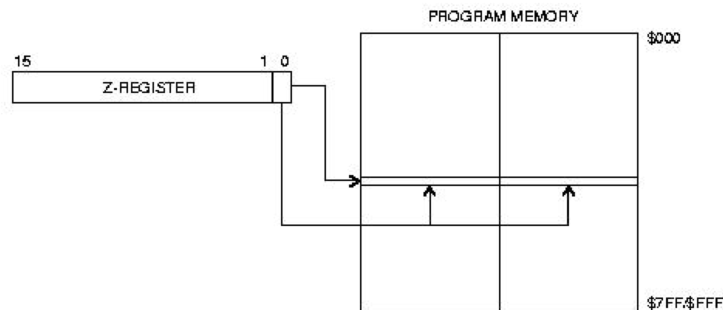

Figura 2.15: *Endereçamento Constante da Memória de Programa*

O conteúdo do registrador Z especifica um endereço constante. OS 15 MSBs[4] selecionam uma palavra de endereço (0–4K), o LSB seleciona o byte de baixa ordem se estiver zerado (LSB = 0), ou o byte de alta ordem se estiver setado (LSB = 1).

Exemplo: LPM r2, Z+ *(r2 ← (Z))*

Endereçamento Indireto da Memória de Programa

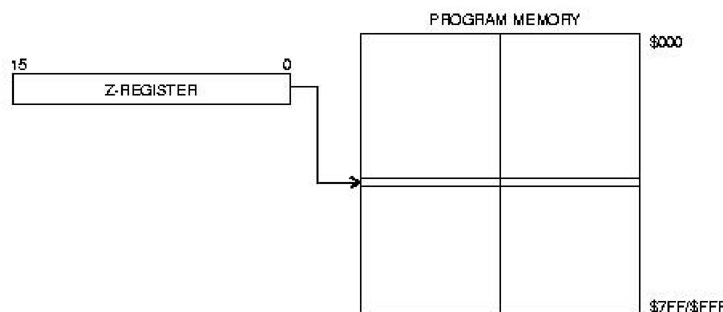

Figura 2.16: *Endereçamento Indireto da Memória de Programa*

A execução do programa continua no endereço contido no registrador Z (i.e o PC é carregado com o conteúdo do registrador Z) Exemplo: IJMP *(PC ← Z(15:0))*

[4]MSBs (Most Significant Bits)

Endereçamento Relativo da Memória de Programa

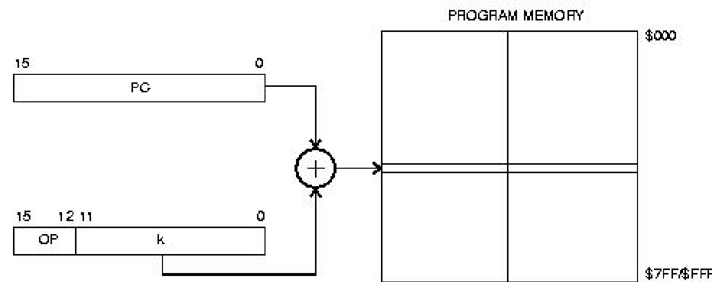

Figura 2.17: *Endereçamento Relativo da Memória de Programa*

A execução do programa continua no endereço PC + k + 1. O endereço relativo k é de -2048 até 2047.

Exemplo: RJMP ok *(desvia o fluxo do código para o label ok)*

2.10 Endereçamento de E/S

Todos os periféricos do 8515 são alocados no espaço de E/S. As posições de E/S são acessadas pelas instruções IN e OUT que transferem dados entre os 32 registradores de uso geral e o espaço de E/S. Os registradores de E/S são acessíveis diretamente bit a bit utilizando as instruções SBI *(Set Bit)* e CBI *(Clear Bit)* dentro do espaço de endereços de $00 a $1F. Nestes registradores, o valor de bits dentro dos registradores pode ser verificado utilizando as instruções SBIS *(Set Bit If Set)* e SBIC *(Set Bit If Clear)*. Quando utilizando as instruções IN, OUT, específicas de E/S, os endereços de E/S $00 até $3F devem ser utilizados. Quando endereçar os registradores de E/S como SRAM, $20 deve ser adicionado a este endereço. O espaço de endereçamento de E/S do 8515 é mostrado na tabela 2.1:

Nota: Posições reservadas e posições não utilizadas não são listadas na tabela. Os endereços indicados entre parênteses são os endereços equivalentes na SRAM.

Endereço	**Nome**	**Função**
$3F ($5F)	SREG	Status Register
$3E ($5E)	SPH	Stack Pointer High
$3D ($5D)	SPL	Stack Pointer Low
$3B ($5B)	GIMSK	General Interrupt Mask register
$3A ($5A)	GIFR	General Interrupt Flag Register
$39 ($59)	TIMSK	Timer/Counter Interrupt Mask register
$38 ($58)	TIFR	Timer/Counter Interrupt Flag register

Endereço	**Nome**	**Função**
$35 ($55)	MCUCR	MCU General Control Register
$33 ($53)	TCCR0	Timer/Counter0 Control Register
$32 ($52)	TCNT0	Timer/Counter0 (8bit)
$2F ($4F)	TCCR1A	Timer/Counter1 Control Register A
$2E ($4E)	TCCR1B	Timer/Counter1 Control Register B
$2D ($4D)	TCNT1H	Timer/Counter1 High Byte
$2C ($4C)	TCNT1L	Timer/Counter1 Low Byte
$2B ($4B)	OCR1AH	Timer/Counter1 Output Compare Register A High Byte
$2A ($4A)	OCR1AL	Timer/Counter1 Output Compare Register A Low Byte
$29 ($49)	OCR1BH	Timer/Counter1 Output Compare Register B High Byte
$28 ($48)	OCR1BL	Timer/Counter1 Output Compare Register B Low Byte
$25 ($45)	ICR1H	Timer/Counter1 Input Capture Register High Byte
$24 ($44)	ICR1L	Timer/Counter1 Input Capture Register Low Byte
$21 ($41)	WDTCR	Watchdog Timer Control Reset
$1F ($3E)	EEARH	EEPROM Address Register High Byte
$1E ($3E)	EEARL	EEPROM Address Register Low Byte
$1D ($3D)	EEDR	EEPROM Data Register
$1C ($3C)	EECR	EEPROM Control Register
$1B ($3B)	PORTA	Data Register, Port A
$1A ($3A)	DDRA	Data Direction Register, Port A
$19 ($39)	PINA	Input Pins, Port A
$18 ($38)	PORTB	Data Register, Port B
$17 ($37)	DDRB	Data Direction Register, Port B
$16 ($36)	PINB	Input Pins, Port B
$15 ($35)	PORTC	Data Register, Port C
$14 ($34)	DDRC	Data Direction Register, Port C
$13 ($33)	PINC	Input Pins, Port C
$12 ($32)	PORTD	Data Register, Port D
$11 ($31)	DDRD	Data Direction Register, Port D
$10 ($30)	PIND	Input Pins, Port D
$0F ($2F)	SPDR	SPI I/O Data Register
$0E ($2E)	SPSR	SPI Status Register
$0D ($2D)	SPCR	SPI Control Register
$0C ($2C)	UDR	UART I/O Data Register
$0B ($2B)	USR	UART Status Register
$0A ($2A)	UCR	UART Control Register
$09 ($29)	UBRR	UART Baud Rate Register
$08 ($28)	ACSR	Analog Comparator Control And Status Register

Tabela 2.1: Endereços de Memória de E/S

2.11 Portas de E/S (A,B,C,D)

O microcontrolador 8515 tem 4 portas de E/S (I/O) programáveis cada uma contando com 8 circuitos independentes. Estas portas são nomeadas PortA, PortB, PortC e PortD.

Inicialmente listamos as características comuns às 4 portas e a seguir as características exclusivas de cada porta.

1. portas de E/S bidirecional de 8 bits.
2. os pinos das 4 portas possuem as seguintes características:
   - podem prover resistores "*pull-up*" internos, selecionáveis bit a bit.
   - Os buffers de saída podem drenar 20 mA.
   - Os pinos ficarão em tri-state quando o sinal de *reset* estiver ativo.

2.11.1 PortA (PA7..PA0)

Os buffers de saída da porta A podem também, como função auxiliar, acionar diretamente displays de LED's.

A porta A também funciona como um multiplexador de Endereços/Dados Entrada/Saída quando utilizando a SRAM externa.

2.11.2 PortB (PB7..PB7)

A porta B também pode controlar através de suas saídas as seguintes funções:

SCK Saída do clock mestre ou entrada para o clock escravo quando utilizando o canal da interface serial (interface SPI).

MISO Entrada mestre de dados ou saída escrava de dados para o canal de interface serial (interface SPI).

MOSI Saída mestre de dados para a interface serial ou entrada de dados escrava para o canal da interface serial (interface SPI).

SS Seletor de porta escrava de entrada *Slave port select* (`input`).

AIN1 Entrada negativa do comparador analógico.

AIN0 Entrada positiva do comparador analógico.

T1 Fonte do contador/temporizador 1.

T0 Fonte do contador/temporizador 0.

2.11.3 PortC (PC7..PC0)

A porta C também tem como função auxiliar, servir como saída de endereço quando utilizando SRAM externa.

2.11.4 PortD (PD7..PD0)

A porta D também tem funções alternativas tais como:

RXD Linha de entrada da UART

TXD Linha de saída da UART

INT0 Entrada externa da interrupção 0

INT1 Entrada externa da interrupção 1

OC1A saída do contador/temporizador A

WR Sinal de escrita para a memória externa

RD Sinal de leitura para a memória externa

2.12 Registrador de Controle Mestre

O registrador de controle da MCU(*Microcontroller Unit*) contêm bits de controle para as funções gerais da MCU.

Bit 7 - SRE Habilitação da SRAM Externa Quando o bit SRE estiver em 1, os dados da SRAM externa estão habilitados/podem ser acessados e as funções dos pinos AD0-7 (porta A), A8-15 (porta C), WR e RD (porta D) são ativadas como sendo as funções alternativas. O bit SRE sobrepõe qualquer configuração de bits e direção nos registradores de dados. Quando o bit SRE estiver desligado (0), a SRAM externa é desabilitada e as configurações normais dos pinos e direção de dados é utilizada.

Bit 6 - SRW External SRAM Wait State Insere *wait state* nos acessos à SRAM Externa.

Bit 5 - SE Sleep Enable Este bit habilita ou não a entrada no modo de controle de consumo de energia quando a instrução SLEEP é executada.

Bit 4 - SM Sleep Mode Este bit controla o modo de baixo consumo de energia, alternando entre o modo de inatividade (*idle mode*) e o modo de desligamento (*power down*).

Bit 3,2 - ISC11, ISC10 Interrupt Sense Control 1 bit 1 and bit 0 Controle da interrupção externa 1.

Bit 1,0 - ISC01, ISC00 Interrupt Sense Control 0 bit 1 and bit 0 Controle da interrupção externa 0.

2.13 Comparador Analógico

O comparador analógico compara os valores na entrada positiva PB2 (AIN0) e na entrada negativa PB3 (AIN1). Quando a tensão na entrada positiva PB2 (AIN0) é maior que a tensão na entrada negativa PB3 (AIN1), a saída do comparador analógico, ACO é um. A saída do comparador pode ser programada para disparar a função de captura do temporizador/contador.

Além disso, o comparador pode disparar uma interrupção associada ao comparador analógico. O usuário pode selecionar o disparo da interrupção se a saída do comparador transicionar de 0 para 1, de 1 para 0, ou se trocar de estado.

2.14 UART

O 8515 tem uma UART (Universal Asynchronous Receiver and Transmitter) dúplex, cujas principais características são:

- Circuito divisor de freqüências que pode gerar inúmeras taxas de transmissão.
- Filtro de ruído.
- Detecção de erros de sobreescrita e enquadramento.
- Interrupções podem ser disparadas em uma ou mais das seguintes condições:
    1. transmissão ou recepção completas.
    2. registrador de transmissão vazio.

2.15 Interface Serial de Periféricos

A interface serial de periféricos (*Serial Peripheral Interface*) permite transferência de dados em alta velocidade entre o 8515 e periféricos ou entre dispositivos AVR. Entre as características da SPI estão:

- Transmissão síncrona dúplex a 3 fios
- Operação como Mestre ou Escravo
- Quatro taxas de transferência programáveis
- Interrupção de Fim de transmissão
- Flag de proteção de colisão de escrita (*Write Collision Flag Protection*)
- Acordar o processador se este está em modo de espera (somente no modo escravo).

2.16 EEPROM

O 8515 contém 512 bytes de dados de memória EEPROM. Esta é organizada como uma área de dados separada, na qual bytes podem ser lidos e escritos um a um. A EEPROM tem um durabilidade de pelo menos 100.000 ciclos de escrita.

O tempo de escrita é da ordem de 2.5 a 4ms, dependendo da tensão de alimentação (V_{cc}). Um temporizador próprio permite que o software do usuário detecte quando o próximo byte pode ser escrito.

2.17 Unidade de Interrupções

O 8515 possui duas máscaras de interrupção de 8 bits, uma delas específica para os contadores/temporizadores. Quando uma interrupção ocorre, o habilitador global de instruções, bit I é zerado e todas as interrupções são desabilitadas. O bit I é habilitado (em 1) quando uma instrução RETI é executada. A seguir podemos observar a tabela 2.2 que mostra o conteúdo do vetor de interrupções.

Posição	**End. Prog.**	**Fonte**	**Descrição**
1	$000	RESET	External Reset, Power-on Reset and Watchdog Reset
2	$001	INT0	External Interrupt Request 0
3	$002	INT1	External Interrupt Request 1
4	$003	TIMER1 CAPT	Timer/Counter1 Capture Event
5	$004	TIMER1 COMPA	Timer/Counter1 Compare Match A
6	$005	TIMER1 COMPB	Timer/Counter1 Compare Match B
7	$006	TIMER1 OVF	Timer/Counter1 Overflow
8	$007	TIMER0 OVF	Timer/Counter0 Overflow
9	$008	SPI, STC	Serial Transfer Complete
10	$009	UART, RX	UART, Rx Complete
11	$00A	UART, UDRE	UART Data Register Empty
12	$00B	UART, TX	UART, Tx Complete
13	$00C	ANA_COMP	Analog Comparator

Tabela 2.2: Vetor de Interrupções

2.18 Temporizadores/Contadores

O 8515 provê dois temporizadores/contadores de uso geral – um de 8 bits e outro de 16 bits. Os temporizadores/contadores tem divisores de freqüência individuais do mesmo tipo do divisor de freqüência de 10 bits do temporizador. Ambos os temporizadores/contadores podem ser utilizados como temporizadores com um clock interno ou como contadores com um pino de conexão externa que dispara a contagem.

Em ambos os temporizadores, quando o temporizador/contador é gatilhado externamente, o sinal externo é sincronizado com o oscilador da CPU. Para garantir uma amostra apropriada do sinal externo, o tempo mínimo entre duas transições do sinal externo deve ser, pelo menos, um período de clock da CPU interna. O sinal externo é capturado na borda de subida do clock interno.

2.19 Watchdog Timer

O *watchdog timer* é controlado por um clock separado de 1MHz. Este é o valor normal quando temos V_{cc} = 5V. Controlando o divisor de freqüência do contador do *watchdog timer*, é possível ajustar o intervalo de reinício do *watchdog timer*, sendo que a instrução WDR - Watchdog Reset - reinicia o contador do *watchdog timer*. Oito períodos de clock diferentes podem ser selecionados para determinar o período de reinício. Se o período terminar antes de outro sinal de reinício para o *watchdog timer*, o 8515 é re-inicializado e executa a partir do vetor de reinício.

Capítulo 3

Programação do AVR8515

3.1 Conjunto de Instruções do AVR8515

Nas próximas seções encontramos a nomenclatura utilizada na programação do 8515, como também tabelas com as instruções do 8515 classificadas pelo tipo da instrução: lógicas e aritméticas, desvio, transferência de dados e manipulação de bits. Estas tabelas tem como objetivo ser uma referência rápida às instruções do 8515.

3.1.1 Nomenclatura utilizada

Registrador de Status (SREG)

C: indicador de Carry

Z: indicador de Zero

N: indicador de Negativo

V: indicador de *overflow* em operações de complemento de dois

S: $N \oplus V$, para testes com sinal

H: indicador de vai-um entre os bits 3 e 4 (*Half Carry Flag*)

T: bit de transferência utilizado pelas instruções BLD e BST

I: indicador que permite habilitar/desabilitar as interrupções globalmente

Registradores e Operandos

Rd: registrador de destino (e fonte) no bloco de registradores

Rr: registrador fonte no bloco de registradores

R: resultado após a instrução ser executada

K: dado constante

k: endereço constante

b: bit no bloco de registradores ou no registrador de E/S (I/O)

s: bit no registrador de status (SREG)

X,Y,Z: registradores de endereçamento indireto (X= R27:R26, Y=R29:R28, Z=R31:R30)

A: endereço de posição de E/S

q: deslocamento para endereçamento direto (6 bits)

Registradores de E/S (I/O)

RAMPX, RAMPY, RAMPZ: registradores concatenados com os registradores X, Y e Z habilitando endereçamento indireto de todo o espaço de dados em MCU's com mais de 64Kbytes de espaço de dados, e busca de dados com constantes em MCU's com mais de 64Kbytes de espaço de programa.

RAMPD: registrador concatenado com o registrador Z habilitando endereçamento direto de todo o espaço de dados em MCU's com mais de 64Kbytes de espaço de dados.

EIND: registrador concatenado com a instrução habilitando um salto indireto e chamadas a todo o espaço de programa em MCU's com mais de 64Kbytes de espaço de programa.

STACK: pilha para o endereço de retorno e registradores empilhados.

SP: ponteiro para a pilha (STACK).

3.1.2 Conjunto de Instruções

O conjunto de instruções do 8515 está subdividido em 4 tabelas separando as classes de instruções: lógicas e aritméticas, desvio, transferência de dados e manipulação de bits.

Instr.	**Op'ndos**	**Operação**	**Flags Afetadas**	**Descrição**
Instruções Lógicas e Aritméticas				
ADD	Rd, Rr	Rd = Rd + Rr	Z,C,N,V,S,H	Add without Carry
ADC	Rd, Rr	Rd = Rd + Rr + C	Z,C,N,V,S,H	Add with Carry
ADIW	Rd, K	Rd+1:Rd = Rd+1:Rd + K	Z,C,N,V,S	Add Immediate to Word
SUB	Rd, Rr	Rd = Rd - Rr	Z,C,N,V,S,H	Subtract without Carry
SUBI	Rd, K	Rd = Rd - K	Z,C,N,V,S,H	Subtract Immediate
SBC	Rd, Rr	Rd = Rd - Rr - C	Z,C,N,V,S,H	Subtract with Carry
SBCI	Rd, K	Rd = Rd - K - C	Z,C,N,V,S,H	Subtract Immediate with Carry
SBIW	Rd, K	Rd+1:Rd = Rd+1:Rd - K	Z,C,N,V,S	Subtract Immediate from Word
AND	Rd, Rr	Rd = Rd • Rr	Z,N,V,S	Logical AND
ANDI	Rd, K	Rd = Rd • K	Z,N,V,S	Logical AND with Immediate
OR	Rd, Rr	Rd = Rd ∨ Rr	Z,N,V,S	Logical OR
ORI	Rd, K	Rd = Rd ∨ K	Z,N,V,S	Logical OR with Immediate
EOR	Rd, Rr	Rd = Rd ⊕ Rr	Z,N,V,S	Exclusive OR
COM	Rd	Rd = $FF - Rd	Z,C,N,V,S	One's Complement
NEG	Rd	Rd = $00 - Rd	Z,C,N,V,S,H	Two's Complement
SBR	Rd, K	Rd = Rd ∨ K	Z,N,V,S	Set Bit(s) in Register
CBR	Rd, K	Rd = Rd • ($FFh - K)	Z,N,V,S	Clear Bit(s) in Register
INC	Rd	Rd = Rd + 1	Z,N,V,S	Increment
DEC	Rd	Rd = Rd - 1	Z,N,V,S	Decrement
TST	Rd	Rd = Rd • Rd	Z,N,V,S	Test for Zero or Minus
CLR	Rd	Rd = Rd ⊕ Rd	Z,N,V,S	Clear Register
SER	Rd	Rd = $FF	Nenhuma	Set Register
MUL	Rd, Rr	R1:R0 = Rd x Rr (UU)	Z,C	Multiply Unsigned
MULS	Rd, Rr	R1:R0 = Rd x Rr (SS)	Z,C	Multiply Signed
MULSU	Rd, Rr	R1:R0 = Rd x Rr (SU)	Z,C	Multiply Signed with Unsigned
FMUL	Rd, Rr	R1:R0 = (Rd x Rr) ≪ 1 (UU)	Z,C	Fractional Multiply Unsigned
FMULS	Rd, Rr	R1:R0 = (Rd x Rr) ≪ 1 (SS)	Z,C	Fractional Multiply Signed
FMULSU	Rd, Rr	R1:R0 = (Rd x Rr) ≪ 1 (SU)	Z,C	Fractional Multiply Signed with Unsigned

Tabela 3.1: Instruções Lógicas e Aritméticas

Instr.	**Op'ndos**	**Operação**	**Flags Afetadas**	**Descrição**
Instruções de Desvio				
RJMP	k	PC = PC + k + 1	Nenhuma	Relative Jump
IJMP		PC(15:0) = Z, PC(21:16) = 0	Nenhuma	Indirect Jump to (Z)
EIJMP		PC(15:0) = Z, PC(21:16) = EIND	Nenhuma	Extended Indirect Jump to (Z)
JMP	k	PC = k	Nenhuma	Jump
RCALL	k	PC = PC + k + 1	Nenhuma	Relative Call Subroutine
ICALL		PC(15:0) = Z, PC(21:16) = 0	Nenhuma	Indirect Call to (Z)
EICALL		PC(15:0) = Z, PC(21:16) = EIND	Nenhuma	Extended Indirect Call to (Z)
CALL	k	PC = k	Nenhuma	Call Subroutine
RET		PC = STACK	Nenhuma	Subroutine Return
RETI		PC = STACK	I	Interrupt Return
CPSE	Rd, Rr	if (Rd = Rr) PC = PC + 2 ou 3	Nenhuma	Compare, Skip if Equal
CP	Rd, Rr	Rd - Rr	Z,C,N,V,S,H	Compare
CPC	Rd, Rr	Rd - Rr - C	Z,C,N,V,S,H	Compare with Carry
CPI	Rd, K	Rd - K	Z,C,N,V,S,H	Compare with Immediate
SBRC	Rr, b	if(Rr[b]=0) PC = PC + 2 ou 3	Nenhuma	Skip if Bit in Register Cleared
SBRS	Rr, b	if(Rr[b]=1) PC = PC + 2 ou 3	Nenhuma	Skip if Bit in Register Set
SBIC	Rr, b	if(IO[A,b]=0) PC = PC + 2 ou 3	Nenhuma	Skip if Bit in I/O Register Cleared
SBIS	Rr, b	if(IO[A,b]=1) PC = PC + 2 ou 3	Nenhuma	Skip if Bit in I/O Register Set
BRBS	s, k	if(SREG[s]=1) PC = PC + k + 1	Nenhuma	Branch if Status Flag Set
BRBC	s, k	if(SREG[s]=0) PC = PC + k + 1	Nenhuma	Branch if Status Flag Cleared
BREQ	k	if(Z = 1) PC = PC + k + 1	Nenhuma	Branch if Equal
BRNE	k	if(Z = 0) PC = PC + k + 1	Nenhuma	Branch if Not Equal
BRCS	k	if(C = 1) PC = PC + k + 1	Nenhuma	Branch if Carry Set
BRCC	k	if(C = 0) PC = PC + k + 1	Nenhuma	Branch if Carry Cleared
BRSH	k	if(C = 0) PC = PC + k + 1	Nenhuma	Branch if Same or Higher
BRLO	k	if(C = 1) PC = PC + k + 1	Nenhuma	Branch if Lower
BRMI	k	if(N = 1) PC = PC + k + 1	Nenhuma	Branch if Minus
BRPL	k	if(N = 0) PC = PC + k + 1	Nenhuma	Branch if Plus
BRGE	k	if($N \oplus V = 0$) PC = PC + k + 1	Nenhuma	Branch if Greater or Equal, Signed
BRLT	k	if($N \oplus V = 1$) PC = PC + k + 1	Nenhuma	Branch if Less Than, Signed
BRHS	k	if(H = 1) PC = PC + k + 1	Nenhuma	Branch if Half Carry Flag Set
BRHC	k	if(H = 0) PC = PC + k + 1	Nenhuma	Branch if Half Carry Flag Cleared
BRTS	k	if(T = 1) PC = PC + k + 1	Nenhuma	Branch if T Flag Set
BRTC	k	if(T = 0) PC = PC + k + 1	Nenhuma	Branch if T Flag Cleared
BRVS	k	if(V = 1) PC = PC + k + 1	Nenhuma	Branch if Overflow Flag is Set
BRVC	k	if(V = 0) PC = PC + k + 1	Nenhuma	Branch if Overflow Flag is Cleared
BRIE	k	if(I = 1) PC = PC + k + 1	Nenhuma	Branch if Interrupt Enabled
BRID	k	if(I = 0) PC = PC + k + 1	Nenhuma	Branch if Interrupt Disabled

Tabela 3.2: Instruções de Desvio

Instr.	**Op'ndos**	**Operação**	**Flags Afetadas**	**Descrição**
Instruções de Transferência de Dados				
MOV	Rd, Rr	Rd = Rr	Nenhuma	Copy Register
MOVW	Rd, Rr	Rd+1:Rd = Rr+1:Rr	Nenhuma	Copy Register Pair
LDI	Rd, K	Rd = K	Nenhuma	Load Immediate
LDS	Rd, k	Rd = (k)	Nenhuma	Load Direct from data space
LD	Rd, X	Rd = (X)	Nenhuma	Load Indirect
LD	Rd, X+	Rd = (X), X = X + 1	Nenhuma	Load Indirect and Post-Increment
LD	Rd, -X	X = X - 1,Rd = (X)	Nenhuma	Load Indirect and Pre-Decrement
LD	Rd, Y	Rd = (Y)	Nenhuma	Load Indirect
LD	Rd, Y+	Rd = (Y), Y = Y + 1	Nenhuma	Load Indirect and Post-Increment
LD	Rd, -Y	Y = Y - 1,Rd = (Y)	Nenhuma	Load Indirect and Pre-Decrement
LDD	Rd, Y+q	Rd = (Y + q)	Nenhuma	Load Indirect with Displacement
LD	Rd, Z	Rd = (Z)	Nenhuma	Load Indirect
LD	Rd, Z+	Rd = (Z), Z = Z + 1	Nenhuma	Load Indirect and Post-Increment
LD	Rd, -Z	Z = Z - 1,Rd = (Z)	Nenhuma	Load Indirect and Pre-Decrement
LDD	Rd, Z+q	Rd = (Z + q)	Nenhuma	Load Indirect with Displacement
STS	k, Rr	Rd = (k)	Nenhuma	Store Direct to data space
ST	X, Rr	(X) = Rr	Nenhuma	Store Indirect
ST	X+, Rr	(X) = Rr, X = X + 1	Nenhuma	Store Indirect and Post-Increment
ST	-X, Rr	X = X + 1, (X) = Rr	Nenhuma	Store Indirect and Pre-Decrement
ST	Y, Rr	(Y) = Rr	Nenhuma	Store Indirect
ST	Y+, Rr	(Y) = Rr, Y = Y + 1	Nenhuma	Store Indirect and Post-Increment
ST	-Y, Rr	Y = Y - 1, (Y) = Rr	Nenhuma	Store Indirect and Pre-Decrement
STD	Y+q, Rr	(Y+q) = Rr	Nenhuma	Store Indirect with Displacement
ST	Z, Rr	(Z) = Rr	Nenhuma	Store Indirect
ST	Z+, Rr	(Z) = Rr, Z = Z + 1	Nenhuma	Store Indirect and Post-Increment
ST	-Z, Rr	Z = Z - 1, (Z) = Rr	Nenhuma	Store Indirect and Pre-Decrement
STD	Z+q, Rr	(Z+q) = Rr	Nenhuma	Store Indirect with Displacement
LPM		R0 = (Z)	Nenhuma	Load Program Memory
LPM	Rd, Z	Rd = (Z)	Nenhuma	Load Program Memory
LPM	Rd, Z+	Rd = (Z), Z = Z + 1	Nenhuma	Load Program Memory and Post-Increment
ELPM		R0 = (RAMPZ:Z)	Nenhuma	Extended Load Program Memory
ELPM	Rd, Z	Rd = (RAMPZ:Z)	Nenhuma	Extended Load Program Memory
ELPM	Rd, Z+	Rd = (RAMPZ:Z), Z = Z + 1	Nenhuma	Extended Load Program Memory and Post-Incr.
SPM		(Z) = R1:R0	Nenhuma	Store Program Memory
ESPM		(RAMPZ:Z) = R1:R0	Nenhuma	Extended Store Program Memory
IN	Rd, A	Rr = I/O(A)	Nenhuma	In From I/O Location
OUT	A, Rr	I/O(A) = Rr	Nenhuma	Out to I/O Location
PUSH	Rr	STACK = Rr	Nenhuma	Push Register on Stack
POP	Rd	Rd = STACK	Nenhuma	Pop Register from Stack

Tabela 3.3: Instruções de Transferência de Dados

Instr.	**Op'ndos**	**Operação**	**Flags Afetadas**	**Descrição**
		Instruções de Manipulação de Bits		
LSL	Rd	Rd[n+1] = Rd[n], Rd[0] = C, C = Rd[7]	Z,C,N,V,H	Logical Shift Left
LSR	Rd	Rd[n] = Rd[n+1], Rd[7] = 0, C = Rd[0]	Z,C,N,V	Logical Shift Right
ROL	Rd	Rd[0] = C, Rd[n+1] = Rd[n], C = Rd[7]	Z,C,N,V,H	Rotate Left Through Carry
ROR	Rd	Rd[7] = C, Rd[n] = Rd[n+1], C = Rd[0]	Z,C,N,V	Rotate Right Through Carry
ASR	Rd	Rd[n] = Rd[n+1], n = 0..6	Z,C,N,V	Arithmetic Shift Right
SWAP	Rd	Rd[3..0] ⇄ Rd[7..4]	Nenhuma	Swap Nibbles
BSET	s	SREG[s] = 1	SREG[s]	Flag Set
BCLR	s	SREG[s] = 0	SREG[s]	Flag Clear
SBI	A, b	I/O[A,b] = 1	Nenhuma	Set Bit in I/O Register
CBI	A, b	I/O[A,b] = 0	Nenhuma	Clear Bit in I/O Register
BST	Rr, b	T = Rr[b]	T	Bit Store from Register to T
BLD	Rd, b	Rd[b] = T	Nenhuma	Bit load from T to Register
SEC		C = 1	C	Set Carry
CLC		C = 0	C	Clear Carry
SEN		N = 1	N	Set Negative Flag
CLN		N = 0	N	Clear Negative Flag
SEZ		Z = 1	Z	Set Zero Flag
CLZ		Z = 0	Z	Clear Zero Flag
SEI		I = 1	I	Global Interrupt Enable
CLI		I = 0	I	Global Interrupt Disable
SES		S = 1	S	Set Signed Test Flag
CLS		S = 0	S	Clear Signed Test Flag
SEV		V = 1	V	Set Two's Complement Overflow
CLV		V = 0	V	Clear Two's Complement Overflow
SET		T = 1	T	Set T in SREG
CLT		T = 0	T	Clear T in SREG
SEH		H = 1	H	Set Half Carry Flag in SREG
CLH		H = 0	H	Clear Half Carry Flag in SREG
NOP		Não faz Nada!!	Nenhuma	No Operation
SLEEP			Nenhuma	Sleep
WDR			Nenhuma	Watchdog Reset

Tabela 3.4: Instruções de Manipulação de Bits

3.2 Material Adicional

Outros trabalhos utilizando o 8515 são muito difundidos como o gcc-avr (ftp://gatekeeper.dec.com/pub/GNU/gcc/gcc-3.0.1/gcc-core-3.0.1.tar.gz) que é uma versão do gcc para a arquitetura AVR. Um trabalho muito interessante é o trabalho: "A GNU development environment for AVR microcontroller" que pode ser encontrado em http://users.rcn.com/rneswold/avr/.

Informações adicionais sobre o ATMEL 8515 e outros microcontroladores podem ser encontradas em:

- Microcontroladores AVR – http://tlw.com/bryce/robot/avr/
- Página do projeto gcc-avr – http://home.overta.ru/users/denisc/
- Lista de discussão do projeto gcc-avr – http://www.avr1.org/mailman/listinfo/avr-gcc-list/
- Como compilar e instalar as ferramentas AVR para o Linux – http://www.isi.edu/~weiye/system/guide/avrtools.html
- Microcontroladores em geral – http://microcontroller.com/default.asp
- Microcontroladores (em alemão) – http://www.mikrocontroller.net/links.htm
- AVR Freaks – http://www.avrfreaks.net/index.php

Capítulo 4

L@V@ - Um Simulador para o 8515

4.1 Descrição Geral do Simulador

O L@V@ é uma ferramenta com duas funções principais: (1) auxiliar o aprendizado de programação em assembly do AVR, e (2) uso no desenvolvimento de software em aplicações do AVR. O L@V@ foi desenvolvido utilizando a linguagem de programação C padrão ansi, a biblioteca ncurses para a interface e Linux como sistema operacional.

O L@V@ permite ao programador o acesso ao conteúdo dos registradores, da pilha e das memórias de programa e dados, o que lhe permite conferir os resultados da execução do programa passo a passo, e encontrar eventuais erros mais facilmente.

Além disso, ao fornecer uma interface "concreta" para o programador, o L@V@ diminui o tempo de aprendizado, pois os erros podem ser verificados e corrigidos mais facilmente em qualquer parte do código.

Outro fator que deve ser levado em consideração é a redução dos custos (mais programadores poderão ser treinados sem que haja custo adicional com hardware) e a eficiência do trabalho final (que aumenta com a qualidade do código produzido).

4.1.1 Interface do Simulador

O L@V@ tem seu fluxo de execução dividido em funções que respondem por partes de tarefas executadas no microcontrolador. Por exemplo, busca, decodificação e execução das instruções são funções existentes no simulador que simulam o comportamento do 8515.

A figura 4.1 mostra a interface de usuário do simulador L@V@. Observa-se na primeira janela o conteúdo da pilha na posição indicada pelo índice (entre parênteses) e o conteúdo das

3 posições antecedentes.

Abaixo da pilha, é mostrado o conteúdo do apontador da pilha utilizado pela instrução corrente (quando aplicável), o contador de programa atual e o número de ciclos já executados.

A seguir, é mostrado o conteúdo dos registradores de uso geral e também dos registradores duplos (X,Y e Z). Ao lado dos registradores de uso geral, é mostrado o conteúdo do registrador de status.

Todos estes valores são mostrados na base hexadecimal.

```
[Stack | Pilha]                                                                  (./.)
(00002) 000000000
[SP]                          [PC]                          [Clock]
000000000004000               000000000004097               000000000000015
[Regs]                                                                  [SREG]
r0: 00  r4: 00  r8 : 00 r12: 00 r16: 33 r20: 21 r24: 00 r28: 00          I: 0  V: 0
r1: 00  r5: 00  r9 : 00 r13: 00 r17: FF r21: 00 r25: 00 r29: 00          T: 0  N: 0
r2: 00  r6: 00  r10: 00 r14: 00 r18: 01 r22: 33 r26: 00 r30: 00          H: 0  Z: 1
r3: 00  r7: 00  r11: 00 r15: 00 r19: 00 r23: 00 r27: 00 r31: 00          S: 0  C: 1
        rX: 0000        rY: 0000        rZ: 0000
[Contexto]                                                                       (//*)
LDI     Rd 16 Rr  0 Kazao   44 Azao    0 q  0 a  0 b  0 s  0 k  0 SP   FA0 OPCODE 0x0D4F PC 17
ADD     Rd 20 Rr 15 Kazao    0 Azao    0 q  0 a  0 b  0 s  0 k  0 SP   FA0 OPCODE 0x2F04 PC 18
MOV     Rd 16 Rr 20 Kazao    0 Azao    0 q  0 a  0 b  0 s  0 k  0 SP   FA0 OPCODE 0x96CE PC 19
AND     Rd 20 Rr 15 Kazao    0 Azao    0 q  0 a  0 b  0 s  0 k  0 SP   FA0 OPCODE 0xFFFF PC 22
[Program Memory]                                                                 (+/-)
 (4097)  2044 2020 2020 6452
 (4101)  3220 2030 7252 3120
 (4105)  2035 614B 617A 206F
 (4109)  2020 3020 4120 617A
 (4113)  206F 2020 3020 7120
 (4117)  2020 2030 2061 3020
 (4121)  6220 2020 2030 2073
 (4125)  3020 6B20 2020 2030
 (4129)  5053 2020 4620 3041
 (4133)  4F20 4350 444F 2045
 (4137)  7830 4646 4646 5020
 (4141)  2043 3232 0000 0000
 (4145)  0000 0000 0000 0000
 (4149)  0000 0000 0000 0000
 (4153)  0000 0000 0000 0000
 (4157)  0000 0000 0000 0000
 (4161)  0000 E7F8 0806 E810
 (4165)  0806 E828 0806 E840
 (4169)  0806 E858 0806 0000
[Teclas de Atalho]
<F1> Help <TAB> Alterna Janelas <q> Quit
```

Figura 4.1: *Interface do Simulador*

Para facilitar a visualização dos valores modificados nos registradores, registradores com valor diferente de zero tem sua cor alterada para: *amarelo, quando registradores de uso geral/duplos* e para *verde, quando registrador de status*.

A janela de contexto é a janela que mostra as últimas 3 instruções executadas, juntamente com a instrução atual, que é a última a ser mostrada (na linha base da janela).

Abaixo da janela de contexto observa-se a *multijanela*. Nesta janela, apertando-se a tecla *TAB*, alterna-se entre as janelas de memória de programa, memória de dados, ajuda e código hexa do arquivo lido respectivamente.

Como última janela visível, observamos a janela de teclas de atalho, onde são mostradas as

principais teclas de atalho e de utilização do L@V@.

Na próxima seção descreveremos o funcionamento da execução das instruções ao longo do simulador.

4.1.2 Fluxo de execução no Simulador

Nota: No decorrer desta seção estaremos apresentando ao lado das estruturas do hardware do 8515 os nomes das estruturas de software correspondentes entre parênteses e em fonte diferenciada.

A execução do simulador ocorre na seqüência:

1. leitura do arquivo .hex (função read_hexa),
2. decodificação das instruções (função decodificador_instrucao),
3. execução das instruções (função execution_step), e
4. modificação dos registradores e exibição na tela.

Inicialmente o simulador é chamado na linha de comando, com um argumento que deve ser um arquivo de formato Intel Hexa.

> O formato Intel Hexa é o formato utilizado para carregar os programas executáveis na memória do microcontrolador e por isso foi escolhido para ser lido pelo simulador. Uma vez que o programa esteja a contento, o programador pode transferir o arquivo direto para o microcontrolador e terá sua aplicação executando no hardware.

A seguir, o simulador inicializa suas estruturas de dados, que incluem o registrador de status (*sreg*), o bloco de registradores (*breg*), memória SRAM (*mem*), memória de programa (*program_memory*), PC (*pc*), SP (*sp*), pilha (*STACK*), estrutura de instrução atual (*instruct*) e estrutura de instruções anteriores (*inst_ant*).

> A *estrutura de instruções anteriores*, que não existe no 8515, é uma estrutura do simulador que mantém o contexto atual da execução do programa, armazenando as 10 últimas instruções executadas.
>
> Todas as estruturas no simulador são inicializadas com zero, com exceção da memória de programas, que é inicializada com 0xFFFF. Este procedimento é adotado para garantir que valores conhecidos sejam atribuídos aos recursos do processador.

Após a inicialização das estruturas, o arquivo .hex recebido como argumento é lido e sua sintaxe é verificada. Caso sua estrutura não seja a esperada, uma mensagem de erro é exibida e o simulador é terminado com estado de saída 1[1].

Se o arquivo está corretamente estruturado, ele é lido, decodificado e seu conteúdo é carregado na memória de programa (*program_memory*).

Após o preenchimento da memória de programa, inicia-se o laço principal do simulador, que em uma volta executa as tarefas de *Decodificação da Instrução, Execução da Instrução Decodificada e Exibição do Estado da Computação*.

Como os programas escritos para o 8515 freqüentemente não terminam, a maneira mais adequada de se finalizar a execução do simulador é pressionando a tecla **q** (*quit*).

4.2 Estruturas utilizadas para implementar o simulador

Esta seção descreve as estruturas de dados usadas na implementação do L@V@. As estruturas de dados a seguir são utilizadas para representar os tipos providos pelo hardware do 8515 e estas não devem ser confundidas com os dados reais do programa simulado.

```
typedef unsigned char tbyte;
typedef unsigned char tdados;
typedef unsigned long int tpcender;
typedef unsigned short int tinst;
typedef unsigned short int tender;
typedef unsigned short int tspender;
```

O tipo *tbyte* representa um tipo de dado cujo tamanho é de 1 byte sem sinal, sendo utilizado em operações que envolvem este tipo (byte) de informação.

O tipo *tdados* tem o mesmo tamanho do tipo anterior, porém, no simulador é utilizado para indicar que a estrutura definida com este tipo (tdados) irá carregar dados do programa e conseqüentemente será utilizada em operações que envolvam a manipulação de dados. Um exemplo é a memória de dados, que é definida utilizando-se este tipo de dados.

Esta distinção entre os diferentes tipos de dados existe para que haja uma maior correlação entre o hardware e o simulador, pois através da utilização desta nomenclatura sabemos em que circuito (de dados/instruções/etc) o sinal estaria ativo no hardware.

Da mesma maneira que existem diferentes circuitos de controle e transmissão no hardware, temos diferentes tipos de dados no simulador, que refletem a utilização destas linhas/circuitos.

[1]Como produzido por `exit(1)`.

Como exemplo, temos o tipo *tpcender* que representa no simulador a linha de dados referente ao PC no hardware do 8515, assim como o tipo *tinst* representa os circuitos de transmissão de dados das instruções.

O tipo *tender* refere-se aos sinais de endereço e suas características.

O tipo *tspender* refere-se aos endereços armazenados na pilha de chamadas de funções.

4.2.1 PC

O contador de programa do 8515 é implementado no simulador através da variável "PC", que é do tipo:

```
tpcender PC;
```

4.2.2 Implementação das Instruções

Na implementação das instruções foram necessárias duas definições de tipos, uma para o *OPCODE* da instrução, e outra que reflete todas as possíveis codificações de uma instrução. Como a codificação das instruções não é regular isso complica a interpretação das instruções, durante a fase de decodificação.

O tipo definido para o *OPCODE* da instrução é:

```
typedef unsigned short int tinst;
```

A estrutura de dados que descreve os campos da instrução contém os seguintes elementos:

Instrucao o opcode *limpo* da instrução[2].

Rd o registrador de destino/fonte (quando aplicável)

Rr o registrador de fonte (quando aplicável)

Kazao constante de Dados.

Azao endereço do posição de E/S

q deslocamento para endereçamento direto

b Bit no bloco de registradores ou no registrador de E/S (I/O)

[2]O opcode padrão, sem as informações de dados.

s Bit no registrador de status (SREG)

k endereço constante

A estrutura é assim definida:

```
typedef struct INSTRUCTION {
  tinst instrucao;
  tbyte Rd;
  tbyte Rr;
  tbyte Kazao;
  tbyte Azao;
  tbyte q;
  tbyte b;
  tbyte s;
  tender k;
}instruction_decoded;
```

Endereços e o apontador de pilha são inteiros de 16 bits.

4.2.3 Apontador da Pilha / Stack Pointer

O tamanho máximo da pilha foi definido como 64Kbytes, uma vez que é possível alocar a pilha na memória SRAM e com a utilização de uma SRAM externa, até 64Kbytes de espaço podem ser utilizados.

```
  #define TAMSTACK 65536
  tspender SP;
  tspender STACK[TAMSTACK];
```

4.2.4 Registradores de Uso Geral

A estrutura do bloco de registradores é definida de modo que o acesso aos registradores possa ser feito através de um registrador simples ou também um registrador duplo (X,Y,Z) em um único acesso.

```
typedef union REG /* tipo de cada um dos registradores */
{
  tbyte reg[32];
  tinst duplo[16];
} bloco_registradores;
```

No simulador o bloco de registradores é definido como:

```
bloco_registradores breg;
```

O acesso aos registradores duplos X, Y e Z que é efetuado com maior freqüência, ocorre através de macros que determinam suas posições no vetor `breg->duplo[ ]`.

Registradores Duplos X Y e Z

```
#define regX 13
#define regY 14
#define regZ 15
```

Acesso aos Registradores

O acesso aos registradores é feito através do bloco de registradores definido. Os registradores simples são acessados diretamente, como podemos observar no excerto de código da execução da instrução *INC*:

```
 breg->reg[Rd] = breg->reg[Rd] + 1;
```

Os registradores duplos também podem ser acessados diretamente, porém com uma sintaxe um pouco diferente, como podemos observar a seguir:

```
 breg->duplo[regX]++;
```

Note-se que a diferença básica entre o acesso aos registradores simples e aos registradores duplos é de `reg` para `duplo` e a utilização das macros (*regX, regY e regZ*) para referenciar os principais registradores duplos.

4.2.5 Registrador de Status

O registrador de status é definido por variáveis separadas para facilitar sua manipulação e o acesso aos bits de status individuais.

```
typedef struct SREG{
  char I,T,H,S,V,N,C,Z;
} status_register;
```

Dentro do simulador definimos o registrador de status como:

```
status_register sreg;
```

4.2.6 Memória de Programas

A memória de programa do 8515 consiste de um bloco de memória de 4Kx16, e é representada pela estrutura mostrada a seguir.

```
typedef struct PROGRAM_MEMORY {
  tinst mem[4096]; /* A memória de programa é composta de 4Kx16 bytes */
}program_memory;
```

Para evitar confusão quanto ao tipo, na declaração foi utilizado o tipo da estrutura como tipo da memória de programa.

```
  struct PROGRAM_MEMORY program_memory;
```

4.2.7 Memória de Dados

A memória de dados do 8515 consiste de um bloco de memória de 512x8, é representada pela estrutura bloco_memoria. Os membros `acessoduplo` e `acessodireto` são usados para implementar os modos de endereçamento dos registradores como se fossem posições de memória de dados. O membro `memnormal` representa a memória de dados presente no hardware.

```
typedef union MEM{
    tdados memnormal[512]; /* a memória do microcontrolador é de 512x8 by-
tes */
    union memduplo acessoduplo;
    union memsimples acessodireto;
}bloco_memoria;

bloco_memoria mem;
```

O conteúdo dos registradores pode ser acessado como em registradores (*add r5, r6*) ou como posições de memória (*add r5, $06*[3]). As estruturas de dados implementadas simulam este comportamento.

[3]Este tipo de acesso não foi testado ainda no simulador.

4.2.8 SRAM

```
#define TAM_MEMORY   65536
```

O tamanho máximo da SRAM no 8515 é de 64Kbytes quando é utilizada memória SRAM externa. O acesso à memória externa não foi implementado no simulador.

4.2.9 Memória de E/S

Dentro do simulador definimos a memória de E/S como:

```
typedef struct IO_MEMORY {
  tinst mem[64];
}io_memory;
```

4.3 Funções do Simulador

Esta seção descreve as funções implementadas no simulador e aponta suas equivalências no hardware do 8515.

O simulador tem sua estrutura central baseada nas funções:

```
int read_hexa(int argc, char *argv[], program_memory *pmem);

struct INSTRUCTION decodificador_instrucao (tinst instrucao);

int execution_step(program_memory memoria_programa, \
instruction_decoded instrucao, status_register *sreg, \
bloco_registradores *breg, bloco_memoria *mem, \
tpcender *PC, tspender *SP, tspender STACK[]);
```

Descreveremos em detalhes o funcionamento de cada uma destas funções nas próximas seções.

4.3.1 Função *read_hexa()*

O objetivo da função read_hexa é ler o arquivo de entrada fornecido ao simulador, verificá-lo, e preencher a memória de programa com as instruções presentes no arquivo.

O formato de arquivo Intel Hexa utiliza uma combinação low-byte, high-byte para expressar as instruções guardadas nos registros de dados. Cada registro começa com um prefixo de 9 caracteres e termina com um checksum de 2 caracteres, tendo o seguinte formato:

```
:BBAAAATTHHHH....HHHCC
```

onde

BB é um número hexadecimal de dois dígitos que representa o número de bytes de dados;

AAAA é um número hexadecimal de quatro dígitos que representa o endereço de início do registro de dados;

TT é um registro de dois dígitos que pode indicar:

- 00 - Registro de Dados
- 01 - Fim de arquivo
- 02 - Registro de Endereço de Continuação de Segmento (*Extended segment address record*)
- 03 - Registro de Endereço de Início de Segmento (*Start segment address record*)

HH é um número hexadecimal que representa uma palavra de dados apresentada como uma combinação "low-byte", "high-byte";

CC é um número hexadecimal que representa o checksum do registro, sendo criado a partir do complemento de dois da soma de todos os bytes precedentes no registro, incluindo o prefixo, ou seja, a soma de todos os bytes + checksum = 00.

Seguindo a convenção acima, observa-se nos arquivos hexa o seguinte:

Cabeçalho padrão:

```
:020000020000FC
```

Fim de arquivo:

```
:00000001FF
```

A função read_hexa lê o cabeçalho padrão, os bytes de prefixo, inverte os dois bytes da palavra de dados, e os copia para a memória de programa. Os bytes da palavra de dados devem ser invertidos porque o padrão Intel Hexa prevê essa inversão durante a criação do arquivo .hex.

Por exemplo, a instrução INC, cujo opcode padrão é 9403, é armazenada dentro do registro de dados no arquivo .hex como 0394. Sendo assim, é necessária a inversão da ordem destes dois bytes para que o opcode fique no formato padrão.

4.3.2 Função *decodificador_instrucao()*

Esta função preenche a estrutura de dados vista na seção 4.2.2 que contêm informações sobre a instrução decodificada. Para facilitar a implementação da decodificação, as instruções foram agrupadas de acordo com as possíveis variações no seu opcode primário/padrão. O *opcode primário* ou *opcode padrão* é o opcode da instrução **sem** os dados da instrução. Os grupos de instruções são mostrados nas tabelas de A.1 a A.11 e contém 116 instruções distintas. Uma listagem completa dos opcodes das instruções, pode ser encontrada no manual [2].

O termo *máscara de bits modificáveis* significa que os bits indicados naquela máscara variam de acordo com o conteúdo dos dados presente na instrução. Um exemplo é a instrução SER que tem a máscara de bits 0x00F0: isto significa que os bits de 8 a 5 do opcode da instrução podem variar dependendo do conteúdo dos dados da instrução. No opcode da instrução SER (em binário): [1110 1111 dddd 1111] podemos observar que os bits "d" são variáveis. Dessa maneira podemos ver que a máscara de bits modificáveis reflete apenas os bits que são variáveis no opcode da instrução.

4.3.3 Função *execution_step()*

A função *execution_step* executa as operações que a instrução decodificada pela função *decodificador_instrucao* determina. O campo *instrucao* da estrutura de dados *instruct* contêm o opcode primário da instrução a ser executada. É com base neste campo que é decidido qual conjunto de operações será executado.

Esta função foi implementada como um seletor em que, a partir da decisão de qual instrução será executada, o conjunto de operações correspondente será selecionado e executado.

Como o conjunto de instruções é quase-regular e existem variações nos tipos das instruções temos as seguintes categorias de instruções:

- Instruções regulares;
- Instruções que tem tamanho de duas palavras; e
- Instruções que precisam saber o tamanho da próxima instrução (no caso de um pulo/desvio).

Instruções regulares são a grande maioria das instruções, que tem tamanho de uma palavra e não necessitam de informações adicionais sobre as próximas instruções. Por exemplo, a instrução *INC*, tem o seguinte conjunto de operações que são executadas:

```
case O_INC:
```

```
{
  breg->reg[Rd] = breg->reg[Rd] + 1;

  R = breg->reg[Rd];

  sreg->V = ((lb(R, Mbit7) >> 7)    &  (~(lb(R, Mbit6) >> 6)) & \
            ((~(lb(R, Mbit5) >> 5)) &  (~(lb(R, Mbit4) >> 4)) & \
            ((~(lb(R, Mbit3) >> 3)) &  (~(lb(R, Mbit2) >> 2)) & \
            ((~(lb(R, Mbit1) >> 1)) &  (~(lb(R, Mbit0)))))));

  sreg->N =  (lb(R,Mbit7) >> 7);

  sreg->S = sreg->N ^ sreg->V;

  sreg->Z = ((~(tb(R,7))) &&  (~(tb(R,6))) && \
            ((~(tb(R,5))) &&  (~(tb(R,4))) && \
            ((~(tb(R,3))) &&  (~(tb(R,2))) && \
            ((~(tb(R,1))) &&  (~(tb(R,0)))))));
  (*PC)++;
  break;
}
```

As instruções que tem tamanho de duas palavras tem um tratamento diferenciado, pelo fato de necessitarem de mais uma palavra da memória para completar a instrução. Listamos aqui as instruções que tem 2 palavras de comprimento:

- CALL
- JMP
- LDS
- STS

Um exemplo de como é executada uma instrução dessa classe é a instrução *JMP* cujo código pode ser visto a seguir:

```
case O_JMP:
  {
    end_jump = ((instruct.k << 16) | memoria_programa.mem[*PC + 1]);
    (*PC) = end_jump;
    break;
  }
```

Observe o trecho do código onde a segunda palavra da instrução é retirada da memória (memoria_programa.mem[*PC + 1]). Em cada uma das instruções listadas acima é possível encontrar esse mesmo trecho de código.

A última classe de instruções, as instruções que precisam conhecer o tamanho da próxima instrução para poder determinar o tamanho do pulo/desvio são as instruções:

- CPSE
- SBRS
- SBRC
- SBIS
- SBIC

Estas instruções fazem (ou não) um salto sobre a próxima instrução dependendo do valor de um indicador ou de um registrador. Podemos observar o código da instrução *CPSE* a seguir:

```
case O_CPSE:
  {
    if (breg->reg[Rd] == breg->reg[Rr]) {
      prox_inst=decodificador_instrucao(memoria_programa.mem[*PC +1]);
      switch (prox_inst.instrucao){
        /* instruções de 2 palavras */
      case O_CALL:
      case O_JMP:
      case O_LDS:
      case O_STS:
        {(*PC)+=3;}
      default:
        {(*PC)+=2;}
      }
    }
    else{
      (*PC)++;
    }
    break;
  }
```

NOTA: Decidimos por efetuar todas as operações dentro de uma mesma função (execution_step) para facilitar a implementação do simulador e também facilitar a compreensão de qual operação está sendo executada para cada instrução. Em decorrência desta decisão, há código duplicado, porém a compreensão facilitada do código compensa a duplicação.

4.3.4 Depuração

Para facilitar a depuração do simulador, basta habilitar a diretiva

```
#define DEBUG
```

no arquivo *sim.h* e recompilar o simulador para que uma saída detalhada de cada passo do simulador seja mostrada na tela.

ATENÇÃO: A saída será muito GRANDE. Tome cuidado com essa opção!

Utilizando a interface curses, a cada instrução executada, o simulador espera que uma tecla seja pressionada para continuar a execução, permitindo assim um total controle da quantidade de instruções executadas.

4.4 Requisitos do Simulador

Sistema Linux com a biblioteca curses instalada.

Terminal X com 110 colunas x 80 linhas (recomendado).

Para gerar o arquivo .hex é recomendada a utilização do tavrasm [4] que é um montador *GPL* que executa sob linux e windows.

Capítulo 5

Conclusão

Durante a fase inicial do projeto, um estudo detalhado sobre o 8515 se fez necessário para que pudéssemos identificar os componentes do hardware do microcontrolador e estimar uma relação com as funções/partes do simulador que seriam desenvolvidas.

Esta fase de estudo consumiu consideráveis horas uma vez que não existe uma especificação de como o microcontrolador é contruído em detalhes. Como uma das finalidades do trabalho consistia em criar um paralelo entre o hardware existente e o software do simulador, alguns pontos tiveram que ser pressupostos. Após identificados os componentes e os circuitos presentes no 8515, o próximo passo do trabalho foi projetar as estruturas que representam os componentes do hardware no simulador.

Uma vez definida a estrutura inicial de quais funções e quais os tipos de dados empregados nas funções, foi dado início ao processo de codificação. Durante a codificação algumas dúvidas surgiram, e foram devidamente compartilhadas com colegas que, em algumas conversas sugeriram alternativas e possibilidades de código.

Era necessário relembrar a cada instante que o objetivo do simulador seria de auxiliar no estudo de alunos com pouca experiência de programação e que este deveria ter um código o mais simples possível para que pudesse ser compreendido com relativa facilidade. Por outro lado, o simulador deveria ser capaz de executar todas as instruções de um programa escrito para o 8515, o que implica em certo grau de complexidade.

Desta maneira o simulador deveria ser simples e ao mesmo tempo robusto. Tendo isso tudo em mente, o código foi desenvolvido de maneira modular (utilizando funções para as principais funcionalidades). Para facilitar a codificação da interface do simulador, foi escolhida a biblioteca *ncurses*[1] devido à facilidade de programação nesta biblioteca.

[1] `http://www.gnu.org/software/ncurses/ncurses.html`

Conhecimentos Adquiridos Durante o projeto, alguns conhecimentos foram adquiridos, como por exemplo sobre o formato *Intel Hexa*, aprofundamento da linguagem de programação C, a programação integrada com a biblioteca *ncurses*, o gerenciamento de um servidor de CVS (Concurrent Version System). Ainda pôde ser agregado a criação de um modo de edição para o emacs que faz *syntax highlighting* de campos nos arquivos de formato Intel Hexa. Some-se ao conhecimento agregado a pesquisa de dados sobre o 8515 e também a engenharia reversa do hardware do 8515.

Trabalhos Futuros Alguns pontos da implementação do L@V@ necessitam de desenvolvimento adicional. Estes são descritos a seguir.

Interface NCurses A interface *ncurses* ainda não está totalmente terminada, faltando detalhes de integração com o núcleo do simulador. Este é o próximo passo a ser executado no trabalho.

Depuração Modelos mais eficientes de depuração foram planejados inicialmente para o simulador, tais como parada em pontos pré-definidos, avanço até determinado valor de PC e avanço até o retorno de uma função, porém não foram implementados devido ao pouco tempo disponível.

Modificação de Estrutura de Instruções A estrutura de instruções pode abrigar ainda um novo campo chamado clock, que irá armazenar o número de ciclos que a instrução demora para executar. Com esse campo poderia haver um melhor controle sobre o número de ciclos já executados.

Sourceforge Como um dos objetivos definidos no início do projeto, pretendemos colocar o projeto do L@V@ no *sourceforge* para que mais pessoas possam usufruir e também contribuir para o projeto. O projeto inicialmente estará disponível no site `http://lava.lezz.org/`.

Apêndice A

Instruções Reconhecidas

SEC	Set Carry Flag
SEZ	Set Zero Flag
SEN	Set Negative Flag
SEV	Set Overflow Flag
SES	Set Signed Flag
SEH	Set Half Carry Flag
SET	Set T Flag
SEI	Set Global Interrupt Flag
CLC	Clear Carry Flag
CLZ	Clear Zero Flag
CLN	Clear Negative Flag
CLV	Clear Overflow Flag
CLS	Clear Signed Flag
CLH	Clear Half Carry Flag
CLT	Clear T Flag
CLI	Clear Global Interrupt Flag
NOP	No Operation
ICALL	Indirect Call to Subroutine
IJMP	Indirect Jump
RETI	Return from Interrupt
RET	Return from Subroutine
SLEEP	Sleep
WDR	Watchdog Reset
EIJMP	Extended Indirect Jump
EICALL	Extended Indirect Call to Subroutine
ESPM	Extended Store Program Memory
SPM	Store Program Memory
LPM	Load Program Memory
ELPM	Extended Load Program Memory

Tabela A.1: Instruções Absolutas, nenhum bit é modificável

As instruções LPM e ELPM tem variações no seu opcode que as classificam em duas categorias: nenhum bit modificável e instruções que tem a máscara de bits modificável 0x01F0

SER	Set all Bits in Register

Tabela A.2: Instruções que tem a máscara de bits modificáveis 0x00F0

MULSU	Multiply Signed with Unsiged
FMUL	Fractional Multiply
FMULS	Fractional Multiply Signed
FMULSU	Fractional Multiply Signed with Unsiged
BCLR	Bit Clear in SREG
BSET	Bit Set in SREG

Tabela A.3: Instruções que tem a máscara de bits modificáveis 0x0077

MOVW	Copy Register Word
MULS	Multiply Signed
ADIW	Add Immediate to Word
SBIW	Subtract Immediate to Word
CBI	Clear Bit in I/O Register
SBI	Set Bit in I/O Register
SBIC	Skip if Bit in I/O Register is Cleared
SBIS	Skip if Bit in I/O Register is Set

Tabela A.4: Instruções que tem a máscara de bits modificáveis 0x00FF

JMP	Jump
CALL	Long Call to a Subroutine

Tabela A.5: Instruções que tem a máscara de bits modificáveis 0x00F1

LDS	Load Direct from SRAM
STS	Store Direct to SRAM
LD	Load Indirect from SRAM using index X/Y/Z
ST	Store Indirect to SRAM using index X/Y/Z
LDD	Load Indirect from SRAM using Displacement
STD	Store Indirect to SRAM using Displacement
LPM	Load Program Memory
ELPM	Extended Load Program Memory
COM	One's Complement
NEG	Two's Complement
SWAP	Swap Nibbles
INC	Increment
ASR	Arithmetic Shift Right
LSR	Logical Shift Right
ROR	Rotate Right Trough Carry
DEC	Decrement
POP	Pop Register from Stack
PUSH	Push Register on Stack

Tabela A.6: Instruções que tem a máscara de bits modificáveis 0x01F0

BLD	Bit Load from T in Sreg to bit in Register
BST	Bit Store from Bit in Register to T in SREG
SBRC	Skip if Bit in Register is Cleared
SBRS	Skip if Bit in Register is Set

Tabela A.7: Instruções que tem a máscara de bits modificáveis 0x01F7

BRCC	Branch if Carry Cleared
BRCS	Branch if Carry Set
BRNE	Branch if Not Equal
BREQ	Branch if Equal
BRPL	Branch if Plus
BRMI	Branch if Minus
BRVC	Branch if Overflow Cleared
BRVS	Branch if Overflow Set
BRGE	Branch if Greater or Equal (Signed)
BRLT	Branch if Less Than (Signed)
BRHC	Branch if Half Carry Flag is Cleared
BRHS	Branch if Half Carry Flag is Set
BRTC	Branch if the T Flag is Cleared
BRTS	Branch if the T Flag is Set
BRID	Branch if Global Interrupt is Disabled
BRIE	Branch if Global Interrupt is Enabled
BRSH	Branch if Same or Higher (Unsigned – BRCC)
BRLO	Branch if Lower (BRCS)

Tabela A.8: Instruções que tem a máscara de bits modificáveis 0x03F8

BRBC	Branch if Bit in SREG is Cleared
BRBS	Branch if Bit in SREG is Set
CPC	Compare with Carry
CP	Compare
SBC	Subtract with Carry
SUB	Subtract without Carry
ADD	Add without Carry
ADC	Add with Carry
CPSE	Compares Skip if Equal
TST	Test for Zero or Minus
AND	Logical AND
EOR	Exclusive OR
OR	Logical OR
MOV	Copy Register
MUL	Multiply
ROL	Rotate Left trough Carry (Same as ADC Rd,Rd)
LSL	Logical Shift Left (Same as ADD Rd,Rd)
CLR	Clear Regiser (Same as EOR Rd, Rd)

Tabela A.9: Instruções que tem a máscara de bits modificáveis 0x03FF

OUT	Store Register to I/O Port
IN	Load an I/O Port to Register

Tabela A.10: Instruções que tem a máscara de bits modificáveis 0x07FF

CPI	Compare with Immediate
SBCI	Subtract Immediate with Carry
SUBI	Subtract Immediate
ORI	Logical OR with Immediate
ANDI	Logical AND with Immediate
LDI	Load Immediate
SBR	Set Bits in register (Same as ORI)
CBR	Clear Bits in Register (Same as ANDI)
RCALL	Relative Call To Subroutine
RJMP	Relative Jump

Tabela A.11: Instruções que tem a máscara de bits modificáveis 0x0FFF

Referências Bibliográficas

[1] ATMEL Corporation. *AVR AT90S8515 Hardware Book*, abril de 1999. Disponível em http://www.atmel.com/dyn/resources/prod_documents/DOC0841.PDF.

[2] ATMEL Corporation. *AVR Instruction Set*, junho de 1999. Disponível em http://www.atmel.com/dyn/resources/prod_documents/DOC0856.PDF.

[3] John L. Hennessy e David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1997.

[4] Tom Mortensen. Tavrasm - A GNU/Linux assembler for the Atmel AVR series of microcontrollers, setembro de 1998. Disponível em http://www.tavrasm.org/.

[5] Sencer Yeralan e Ashutosh Ahluwalia. *Programming and Interfacing the 8051 microcontroller*. Addison Wesley, 4th edition, 1997.